1 слайд:

Что такое обратная совместимость?
— это свойство операционной системы, программного обеспечения, реального продукта или технологии, которое обеспечивает совместимость (interoperability – характеристика продукта или системы работать с другими продуктами или системами) со старыми легаси-системами или с входными данными, предназначенными для такой системы.

Плюсы обратной совместимости:

- Удобство для пользователей: пользователи могут использовать новое программное или аппаратное обеспечение с уже имеющимся старым оборудованием или программным обеспечением, что позволяет избежать необходимости покупать новое оборудование или обновлять старое.
- Удобство для пользователей: обратная совместимость упрощает жизнь пользователям, позволяя им сохранять свои данные и настройки при переходе на новую версию программного или аппаратного обеспечения.
- Защита инвестиций: обратная совместимость сохраняет ценность уже имеющихся инвестиций в старое оборудование или программное обеспечение, что помогает избежать лишних затрат на замену устаревшего оборудования.
- Уменьшение рисков: обратная совместимость уменьшает риски неудачных инвестиций в новое оборудование или программное обеспечение, которое может не оправдать ожиданий.
- Удобство для бизнеса: обратная совместимость позволяет бизнесу сохранять свою инфраструктуру и процессы без необходимости внесения значительных изменений, что уменьшает риски и экономит деньги.
- Поддержка стандартов: обратная совместимость способствует поддержке стандартов, что повышает совместимость между различными системами и упрощает обмен данными.

Все плюсы имеют исключительно бизнес составляющую:

Не трудно заметить, что большинство плюсов напрямую относится к коммерческой составляющей ПО. И это не удивительно, ведь самое главное здесь – продажи. Наивно требовать от коммерческих компаний следовать правилам, которые противоречат правилам бизнеса. А у бизнеса нет никаких проблем с легаси, пока то не затрудняет дальнейшую разработку и продажи. Оптимальный код или неоптимальный — это их вообще не волнует. Лишь бы продукт продавался и удовлетворял нужды клиентов.

2 слайд:
Что такое технический долг?

Технический долг - накопленные в программном коде или архитектуре проблемы, связанные с пренебрежением к качеству при разработке программного обеспечения и вызывающие дополнительные затраты труда в будущем. Технический долг обычно незаметен для конечных пользователей продукта, а связан с недостатками в сопровождаемости, тестируемости, понятности, модифицируемости, переносимости. По аналогии с финансовым долгом технический долг может обрастать «процентами» — усложнением (или даже невозможностью) продолжения разработки, дополнительным временем, которое разработчики потратят на изменение программного продукта, исправление ошибок, сопровождение и т. п. Хотя увеличение технического долга как правило негативно влияет на будущее проекта, оно может быть и сознательным, компромиссным решением, продиктованным сложившимися обстоятельствами.

Причины:

У появления технического долга может быть много причин – давление бизнеса, отсутствие процессов или понимания, отсутствие документации, взаимодействия в команде, отложенный рефакторинг или отсутствие опыта.
Довольно часто проблема возникает потому, что мы просто не в силах предсказать будущее.

3 слайд:

IPv4 – появление, адресация (классовая, CIDR):

В 1981 году «Интернета» хватало всем и каждому — была описана первая широко используемая версия протокола IPv4, использующая 32-битные адреса, ограничивающие адресное пространство 4 294 967 296 возможными уникальными адресами, что казалось огромным числом по сравнению с количеством компьютеров.

Первый октет обозначал адрес сети, за которым следовал локальный адрес хоста, занимавший оставшиеся три октета. Таким образом, стандарт допускал существование 2^8=256 сетей по 2^24=16 777 216 хостов в каждой.

Однако очень скоро выяснилось, что сетей слишком мало, они слишком большие, и адресация лишена гибкости, из-за чего была введена классовая адресация. Идея заключается в следующем: для гибкости в назначении адресов сетей и возможности использовать большое число малых и средних сетей адресное пространство было разделено на несколько логических групп и в каждой группе отводилось разное соотношение хостов и подсетей. Эти группы носят названия классов сетей и пронумерованы латинскими буквами: A, B, C, D и E. Но даже при таком подходе адреса расходовались неэкономно. Класс с самым малым кол-вом узлов(C) поддерживал 256 их экземпляров, таким образом, маленькие компании, которым нужно было на порядок меньше, все равно получали именно 256 адресов узла, большая часть из которых оставалась неиспользуемой (например, компания может иметь адресное пространство из 254 (2^8-2) адресов и использовать только 25 из них).

Эта проблема была осознанна в 90-х годах. Одним из ее решений стало появление маршрутизаторов бесклассовых доменов Интернета (CIDR). До изобретения CIDR, вы могли получить один из трех размеров сети. После изобретения CIDR стало возможным разбивать сети на подсети.

Так, например, если вам нужно 5 IP адресов, ваш провайдер предоставит вам сеть размером 3 бита, которая даст вам 6 IP адресов. Это позволит вашему провайдеру использовать адреса более эффективно. Несомненно, это помогло, но ограниченность адресов все равно была ощутимой проблемой.

Другие дефекты:

Также IPv4 содержала ряд других дефектов, например, заголовок IPv4 был переменной длины. Это было приемлемо, когда маршрутизация осуществлялась программно. Но теперь маршрутизаторы строятся аппаратно, а обрабатывать заголовки переменной длины в аппаратуре сложно. Еще одна проблема с IPv4 заключается в том, что на момент выделения адресов Интернет был американским изобретением. IP-адреса для остального мира фрагментированы. Необходима была схема, позволяющая объединять адреса по географическому признаку, чтобы таблицы маршрутизации были меньше.

IPv6 и решение проблем:

В 1995 году появилась новая версия стандарта – Ipv6, которая имела 128-битные адреса. Почему их так много? Ответ заключается в том, что разработчики были обеспокоены неэффективной организацией адресов, поэтому существует так много доступных адресов, которые мы можем распределить неэффективно для достижения других целей. Обычно, провайдер получает сетевое пространство в 80 бит. Верхние 48 бит дополнительно разделяются, чтобы провайдеры, находящиеся «близко» друг к другу, имели схожие диапазоны сетевых адресов, что позволяет объединять сети в таблицах маршрутизации.

Заголовок IPv4 имеет переменную длину. Заголовок IPv6 всегда имеет фиксированную длину 40 байт. В IPv4 дополнительные опции приводили к увеличению размера заголовка. В IPv6, если требуется дополнительная информация, она хранится в заголовках расширения, которые следуют за заголовком IPv6 и обычно не обрабатываются маршрутизаторами, а обрабатываются программным обеспечением в месте назначения.

Неполная совместимость:

Проблема заключается в том, что IPv6 разработан без полноценной совместимости с IPv4, узел с поддержкой только IPv6 не может подключиться к узлу, работающему только по IPv4. Переход от IPv4 к IPv6 требовал «двух стековой» фазы, во время которой хост-компьютер взаимодействовал бы с обоими стеками протоколов одновременно, используя стек протокола IPv6 для взаимодействия с другими хост-компьютерами IPv6, а стек протокола IPv4 для взаимодействия с другими хост-компьютерами IPv4.

Вывод: что по итогу? Изначально мы имели IPv4, который на момент создания довольно хорошо исполнял свои функции. Но невозможность предусмотреть все (например, резкое увеличение кол-ва пользователей в сети, изменение способа обработки заголовков с программного на аппаратный, а также необходимость группировать сети по географическому признаку) привели к неудобствам и изобретению новой версии протокола, которая учитывала все эти нюансы, но переход на которую продолжается до сих пор из-за неполной совместимости.

4 слайд:

Как запускается процессор:

Центральные процессоры (CPU) не могут ничего сделать, пока им не скажут, что делать. Возникает очевидная проблема — как вообще заставить CPU что-то делать? Во многих CPU эта задача решается при помощи вектора сброса — жёстко прописанного в CPU адреса, из которого нужно начинать считывать команды при подаче питания. Адрес, на который указывает вектор сброса, обычно представляет собой какую-нибудь ROM или флэш-память, которую CPU может считать, даже если никакое другое оборудование ещё не сконфигурировано. Это позволяет производителю системы создавать код, который будет исполнен сразу же после включения питания, сконфигурирует всё остальное оборудование и постепенно переведёт систему в состояние, при котором она сможет выполнять пользовательский код.

Конкретная реализация вектора сброса в системах x86 со временем менялась, но, по сути, это всегда были 16 байтов ниже верхушки адресного пространства, то есть 0xffff0 на 20-битном 8086, 0xffffff0 на 24-битном 80286 и 0xfffffff0 на 32-битном 80386. По стандарту в системах x86 ОЗУ начинается с адреса 0, поэтому верхушку адресного пространства можно использовать для размещения вектора сброса с минимальной вероятностью конфликта с ОЗУ.

Код прошивки выполняется в real mode:

самое примечательное в x86 здесь то, что когда он начинает выполнять код из вектора сброса, он всё ещё находится в режиме реальной адресации (real mode). Адреса не абсолютны, это 16-битные смещения, прибавляемые к значению, хранящемуся в «сегментном регистре». Для кода, данных и стека существуют собственные сегментные регистры, поэтому 16-битный адрес может относиться к разным реальным адресам в зависимости от того, как он интерпретируется. Всё это нужно для сохранения совместимости со старыми чипами, вплоть до того, что даже 64-битные x86 запускаются в real mode с сегментами и всем остальным (и тоже начинают исполнение с 0xfffffff0, а не с 0xfffffffffffffff0 — 64-битный режим не поддерживает real mode, поэтому 64-битный физический адрес невозможно выразить при помощи сегментных регистров, и мы всё равно начинаем сразу ниже 4 ГБ, даже несмотря на то, что доступно гораздо большее адресное пространство).

5 слайд:

UEFI – прошивка:

В современных системах UEFI-прошивка, запускаемая из вектора сброса, перепрограммирует CPU во вразумительный режим (то есть без всей этой фигни с сегментацией), выполняет различные действия, например конфигурирует контроллер памяти, чтобы можно было получить доступ к ОЗУ (при этом процессе кэш CPU используется как ОЗУ, потому что программирование контроллера памяти — довольно сложная задача, ведь нужно хранить больше информации о состоянии, чем поместится в регистрах, то есть необходимо ОЗУ, но у нас нет ОЗУ, пока не заработает контроллер памяти)

Как все происходит на самом деле:

на самом деле, всё не так. Современный Intel x86 запускается иначе. Всё гораздо страннее. Да, кажется, что именно так всё и происходит, но за кулисами выполняется множество других действий. Давайте поговорим о безопасности запуска. Принцип любого вида верифицируемого запуска (например, UEFI Secure Boot) в том, что подпись на следующем компоненте цепочки запуска валидируется до исполнения компонента. Но что верифицирует первый компонент цепочки запуска? Нельзя просто попросить BIOS проверить саму себя — если нападающий сможет заменить BIOS, то его версия BIOS просто соврёт о том, что это сделано. Intel решила эту проблему с помощью Boot Guard. Но прежде, чем мы доберёмся до Boot Guard, нам нужно обеспечить работу CPU в максимально свободном от багов состоянии. Поэтому при своём запуске CPU исследует флэш-память системы и ищет заголовок, указывающий на обновления микрокода CPU. В комплекте CPU компании Intel есть встроенный микрокод, но часто он старый и забагованный, поэтому прошивка системы может решить включить копию, которая достаточно новая, чтобы надёжно работать. Образ микрокода копируется из флэш-памяти, проверяется подпись и новый микрокод начинает работать. Это справедливо и с использованием Boot Guard, и без него. Однако в случае Boot Guard перед переходом к вектору сброса микрокод в CPU считывает из флэш-памяти Authenticated Code Module (ACM) и проверяет его подпись с прописанным Intel ключом. Если они совпадают, он начинает исполнять ACM. Здесь следует помнить, что CPU не может просто верифицировать ACM, а затем исполнить его непосредственно из флэш-памяти: если бы он сделал это, то флэш-память могла бы распознать это, передавать для верификации подлинный ACM, а затем передать CPU другие команды при их повторном считывании для исполнения (уязвимость Time of Check vs Time of Use, или TOCTOU).

Ну да ладно. Теперь мы загрузили и верифицировали ACM, после чего его можно спокойно исполнить. ACM выполняет разные вещи, но самое важное с точки зрения Boot Guard заключается в том, что он считывает набор фьюзов с однократной записью на

чипсете материнской платы, представляющий собой SHA256 публичного ключа. Затем он считывает первый блок прошивки (Initial Boot Block, или IBB) в ОЗУ (точнее, как говорилось выше, в кэш) и парсит его. В нём есть блок, содержащий публичный ключ — он хэширует этот ключ и верифицирует, что он соответствует SHA256 из фьюзов. Затем он использует этот ключ для валидации подписи на IBB. Если всё правильно, он исполняет IBB, и всё начинает выглядеть как красивая простая модель, о которой мы говорили выше.

6 слайд:

Только не кажется ли вам, что весь этот код чрезвычайно сложно реализовать в real mode? И да, выполнение всех этих расчётов современной криптографии только с 16-битными регистрами было бы очень мучительной задачей. Поэтому всё происходит не так. Всё это происходит в абсолютно вразумительном 32-битном режиме, а после этого CPU на самом деле переключается в ужасную конфигурацию, чтобы сохранить совместимость с 80386, выпущенным в 1986 году.

Разумеется, как говорилось выше, в современных системах прошивка потом перепрограммирует CPU во что-то более вразумительное, поэтому разработчикам ОС больше не нужно об этом волноваться [1][2]. Это значит, что мы перескакивали между несколькими состояниями только из-за вероятности того, что кто-то захочет запустить легаси-BIOS, а затем загрузить DOS на CPU, в котором на пять порядков больше транзисторов, чем в 8086.

7 слайд:

Порча софта винды ради обратной совместимости:

Одержимость обратной совместимостью Windows – это, наверное, одна из причин, почему операционная система стала столь популярной. Все это началось с Windows 95. Руководство MS понимало, что приоритетная задача для принятия публикой новой ОС — это совместимость приложений. Один из разработчиков Microsoft в те годы Раймонд Чен вспоминает забавный факт: чтобы обеспечить максимально широкий охват для тестирования приложений на совместимость, менеджер по разработке Windows 95 сел в свой пикап, поехал в местный магазин Egghead Software (специальные магазины, где продавался коробочный софт) и купил по одной копии всех программ в магазине. Затем он вернулся в Microsoft, выгрузил все коробки на столы в кафетерии и предложил каждому члену команды разработчиков Windows95 прийти и взять по две программы. Основные правила заключались в том, что вы должны установить программу, использовать её как обычный юзер — и составить отчёт на каждый найденный баг, даже самый мелкий… В обмен сотрудник получал право оставить эти программы себе бесплатно после выхода Windows 95.

Известен случай, как в Windows 95 специально внесли баг ради совместимости с популярной игрой SimCity. Джон Росс случайно оставил в SimCity баг с чтением только что освобождённой памяти. Мда… Игра прекрасно работала на Windows 3.x, потому что там освобождённую память никто не занимал, однако в бета-версии Windows 95 она, естественно, падала. И вот тут самое интересное. Microsoft отследила баг и добавила в Windows 95 специальный код, который ищет SimCity. Если находит, то запускает аллокатор памяти в специальном режиме, который не сразу освобождает память. Именно такая одержимость обратной совместимостью заставляла людей охотно переходить на Windows 95.

Эта тенденция продолжилась в последующих версиях Windows. ОС была обязана запускать все программы, которые запускались на предыдущей версии, дополнительно к нововведённым стандартам. Например, для игры Final Fantasy VII в Windows NT был реализован хак под названием Win95VersionLie, который пытался убедить игру в том, что рабочее окружение соответствует Win 95 и содержит необходимые файлы (на самом деле они отсутствуют).

8 слайд:

Архаичные технологии винды:

Но погоня за обратной совместимостью распространяется не только на игры, но и на другие стандарты и протоколы. В итоге — сейчас за Windows тянется целый шлейф архаичных технологий, от которых компания не хочет (или не может) отказаться по требованию клиентов. Среди них:

- запуск древних 16-битных бинарников от Windows 1.0 (на 32-битной ОС) и почти всех приложений Win32 на Win64;
- поддержка старых форматов файлов, включая пакетные файлы;
- древний интерпретатор cmd.exe (от него не могут избавиться, потому что тысячи клиентов за десятилетия внедрили в продакшн миллионы пакетных файлов);

9 слайд:

Сравнение линуха и винды:

Поскольку новые функции добавляют, а старые не убирают, это естественным образом раздувает код. В ядре Linux тоже есть проблемы с этим, но гораздо в меньшей степени. Там Линус Торвальдс всё-таки понимает важность оптимизации и проводит строгую чистку кодовой базы. Например, 21 октября 2022 года он предложил удалить из ядра поддержку процессорной архитектуры 80486 (80386 удалили в 2012-м). Такая оптимизация сократит потребление памяти и увеличит производительность ядра, поскольку в ядро непрерывно приходится вносить разные костыли (workarounds) для поддержки старых CPU.