ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

ЗАЩИТА ИНФОРМАЦИИ

ЛАПОНИНА
ОЛЬГА РОБЕРТОВНА

ВМК МГУ

КОНСПЕКТ ПОДГОТОВЛЕН СТУДЕНТАМИ, НЕ ПРОХОДИЛ ПРОФ. РЕДАКТУРУ И МОЖЕТ СОДЕРЖАТЬ ОШИБКИ. СЛЕДИТЕ ЗА ОБНОВЛЕНИЯМИ НА VK.COM/TEACHINMSU.

ЕСЛИ ВЫ ОБНАРУЖИЛИ ОШИБКИ ИЛИ ОПЕЧАТКИ, ТО СООБЩИТЕ ОБ ЭТОМ, НАПИСАВ СООБЩЕСТВУ VK.COM/TEACHINMSU.

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА
СТУДЕНТКУ ФИЗИЧЕСКОГО ФАКУЛЬТЕТА МГУ
КУЩЕНКО АННУ КОНСТАНТИНОВНУ

ОГЛАВЛЕНИЕ

ЛЕКЦИЯ 1. СЕРВИСЫ БЕЗОПАСНОСТИ

Проблемы, связанные с защитой информации, обеспечением безопасности информации, возникли давно, намного раньше, чем появились компьютеры. Уже давно людям хотелось, чтобы не вся информация была доступна абсолютно всем. Тогда эта проблема решалась закрытыми дверями, замками и охраной. С появлением компьютеров подобные способы защиты никуда не исчезли, но если раньше мы могли полностью защитить информацию с помощью «хорошего замка», то сейчас это не обеспечит стопроцентную защиту. С помощью административных организационных мер или, говоря другими словами, методом «хорошего замка» можно обеспечить лишь 80-90% защиты. Таким образом, наш курс будет посвящен защите оставшихся 10-20% информации.

Информационные ценности

Введем основные понятия, которые будем использовать в курсе.

Собственник – это обладатель информационных ценностей, он решает, что защитить, насколько это ценно для него и для других, захотят ли его вообще взломать, и сколько, грубо говоря, он готов вложить денег в средства информационной безопасности.

Противник или *оппонент* – это тот, кто хочет получить доступ к ценностям *собственника*. Он совершает так называемые *атаки*.

Уязвимость – это слабое место в системе, с помощью которого можно нарушить информационные ценности. Далее мы будем говорить о том, какие уязвимости существуют, и как от них защититься.

Риск – это вероятность того, что данная уязвимость будет использована. Возможна ситуация, когда уязвимость существует, но вероятность того, что она будет использована, крайне мала.

Проанализировав уязвимости, риски и атаки, собственник создает, так называемую, *политику безопасности* – это совокупность организационных, технических, административных мер, с помощью которых производится защита информационных ценностей. Мы будем рассматривать исключительно технические меры, помня о том, что организационные меры никуда не исчезли. Отметим, что под безопасностью подразумевается именно то, что прописано в политике безопасности. Например, если нам важно, чтобы у нас не сканировали порты, мы должны явно

прописать это в политике безопасности и предусмотреть способы защиты, а если нам это не важно, то мы спокойно позволяем сканировать.

Вообще говоря, лучший механизм обеспечения безопасности – это обрезание проводов, то есть мы отключаем компьютер от сети, выключаем и запираем в шкаф. Однако это мало кого устроит, таким образом, мы идем на риск ради удобства пользования. В нашем курсе мы в основном будем заниматься сетевой безопасностью.

Классификация сетевых атак

У нас есть отправитель A и получатель B, их часто называют Алиса и Боб, Алиса передает информацию Бобу. Из курса по сетевым технологиям вызнаете, что информация на самом деле передается в виде отдельных пакетов, которые проходят через достаточно большое количество, так называемых, маршрутизаторов (рис. 1.1). Без них сеть не будет работать, но они могут делать не только то, что они обязаны, но и то, что захотят.

Рис. 1.1. Нормальный информационный поток.

Рассмотрим первый класс атак, так называемые, *пассивные атаки*. В этом случае оппонент находится на канале и просматривает передаваемую информацию (рис. 1.2).

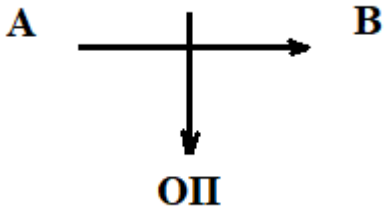

Рис. 1.2. Схема пассивной атаки.

В случае *активной атаки* оппонент уже может что-то сделать с передаваемой информацией. Выделяют несколько видов подобных атак:

- *Модификация* передаваемого сообщения или *man in the middle* (рис. 1.3).

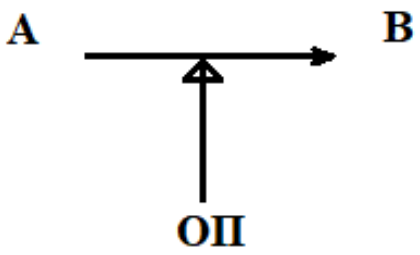

Рис. 1.3. Схема модификации.

- *Фальсификация* (рис. 1.4). А ничего не передает В, а оппонент пытается получить доступ к В от имени А.

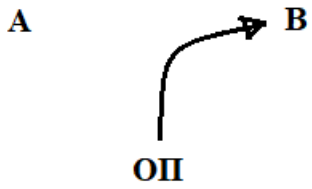

Рис. 1.4. Схема фальсификации.

- *Атаки повтора* или *replay-атаки* (рис. 1.5). Этот вид атак является разновидностью атак фальсификации, его выделяют отдельно, потому что он предполагает определенные методы защиты. А передает В некий блок данных в защищенном виде, то есть оппонент не может его прочитать, также с помощью этого блока данных выполняется аутентификация отправителя А. Оппонент может лишь сохранить этот блок и передать его В с некоторой временной задержкой. Соответственно, у В должна быть возможность отличить блок данных от законного пользователя и от нарушителя.

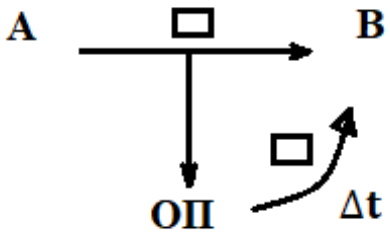

Рис. 1.5. Схема атаки повтора.

- *DOS-атаки* (Deny Of Service) (рис. 1.6). Оппонент пытается либо исчерпать ресурсы в канале, либо исчерпать ресурсы у B так, чтобы законный пользователь не мог получить доступ к B. Исчерпать можно все, что угодно: оперативную память, внешнюю память, процессор, пропускную способность канала, файлы на внешних устройствах, записи в базах данных. Это самый трудно предотвращаемый тип атаки, обычно это связано с маломощными компьютерами, каналами с малой пропускной способностью или с ошибками в используемом программном обеспечении.

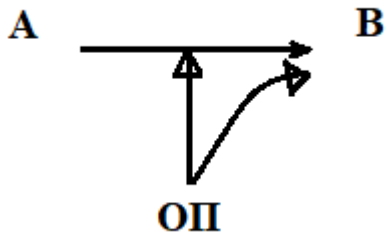

Рис. 1.6. Схемы DOS-атаки.

Первыми DOS-атаками, с котороми столкнулись в эпоху сетевых технологий, были, так называемые, *SYN-атаки*. Напомню из курса по сетевым технологиям, что один из основных протоколов транспортного уровня – это TCP-протокол, который обеспечивает нам надежное соединение, то есть он пересчитывает все пакеты и гарантирует, что ни один не потеряется. Есть фаза установления TCP-соединения: клиент (C) является инициатором и посылает пакет с флагом SYN, сервер (S) отвечает пакетом с флагом SYN ACK, в этот момент он отводит в оперативной памяти место, помечает его как не откачиваемое для того, чтобы там хранить всю информацию о данном соединении. После этого клиент посылает пакет с флагом ACK, и выделенный массив помечается уже как откачиваемый, и в случае необходимости может быть откачен на внешнюю память.

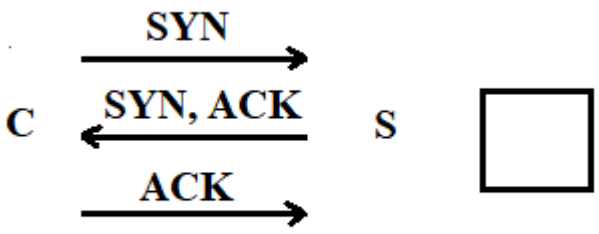

Рис. 1.7. Схема установления TCP-соединения.

Наконец, разберемся, в чем же состоит SYN-атака. Нарушитель высылает пакет с флагом SYN, сервер добросовестно выделяет память и отвечает пакетом с флагом SYN ACK, далее нарушитель присылает не пакет ACK, завершающий установление соединения, а еще один новый пакет с флагом SYN, сервер снова выделяет память. И так продолжается, пока на сервере не исчерпается оперативная память, и сервер не сможет устанавливать соединение ни с нарушителем, ни с законным пользователем. Первые версии операционных систем все были подвержены SYN-атаке, необходимо переписывать операционную систему, чтобы нельзя было бесконечно долго держать полуоткрытые соединения. Заметим также, что сами возвращаемые пакеты SYN ACK нарушителю не нужны, то есть он может устанавливать соединение со случайного IP-адреса, никак не выявляя себя. Современные операционные системы, как правило, не подвержены SYN-атакам.

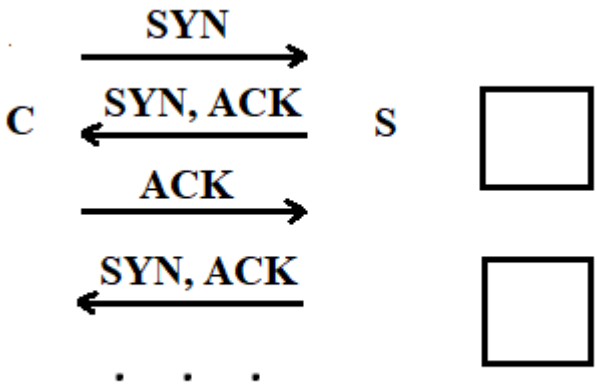

Рис. 1.8. Схема SYN-атаки.

Существуют также *DDOS-атаки* (Distributed Deny Of Service), в этом случае нарушитель создает целую сетку компьютеров, с которых по сигналу атакуется получатель.

Сервисы безопасности

Существуют следующие типы сервисов, с помощью которых можно защититься от сетевых атак:

- *Конфиденциальность* – это защита от пассивной атаки. Мы должны научиться перемешивать передаваемое сообщение таким образом, чтобы никто кроме получателя не смог его прочитать.

- *Целостность* – это возможность получателя определить, что сообщение не было изменено. Мы не можем гарантировать неизменность сообщения, потому что оно передается по различным сетевым устройствам (маршрутизаторы), которые мы не контролируем, однако мы можем определить, было ли сообщение изменено. Возможны ситуации, когда нам не важна конфиденциальность, то есть если мы никаких секретов не передаем, но очень важно, чтобы получатель получил именно то, что ему посылали.

  Рассмотрим пример, у нас есть сервис dns, который выполняет отображение между dns-именами и IP-адресами. Итак, мы набираем в строке браузера dns-имя, клиентский модуль обращается к DNS-серверу с просьбой получить IP-адрес данного dns-имени, сервер его возвращает, далее машина обращается к маршрутизаторам и идет по данному IP-адресу. Если будет выполнена атака man in the middle, то упомянутый адрес будет заменен на адрес нарушителя, и наша машина пойдет по неверному адресу, а что будет дальше можно фантазировать сколько угодно, но точно ничего хорошего. Замети что, ни один из участников взаимодействия не взломан: ни наш компьютер, ни DNS-сервер, ни маршрутизатор, а произошло нарушение целостности.

Рис. 1.9. Схема взлома

- *Доступность* – защита от DOS-атак.

- *Аутентификация* – доказательство того, что участники являются требуемыми, то есть А – это А, В – это В. Если передается единственное сообщение от А к В, то этого достаточно. Если же между А и В устанавливается информационный канал, то есть сообщения передаются в обе стороны, то дополнительно

требуется, чтобы после установления канала нарушитель не мог подстроиться ни под одну из сторон.

- *Авторизация* – предоставление прав доступа, то есть если пользователь А получил доступ к В, это не означает, что ему доступно абсолютно все, а лишь определенное множество ресурсов. Классический пример авторизации, если речь идет о UNIX и файловой системе, то это права доступа чтение-запись-выполнение (rwx) для собственника, аналогично для группы и для всех остальных. Аналогичные права доступа есть и в операционной системе Windows. Права доступа часто называют ACL (Access Control List).

- *Идентификация* отвечает за то, каким образом назначать идентификаторы участникам.

- *Невозможность отказа*. До сих пор мы неявно предполагали, что отправитель и получатель являются дружественными участниками процесса, и опасность исходит извне. Однако, на самом деле, и отправитель, и получатель могут выступать в роли нарушителя. Например, в случае покупателя и продавца: покупатель может хотеть получить неоплаченный товар, а продавец – не отпустить оплаченный товар. В обычной жизни у нас есть подпись, и считается, что, когда мы ставим свою подпись на бумаге, графолог всегда может определить: мы ее подписали или кто-то другой, и к тому же на бумаге ничего нельзя исправить, так как любое исправление будет видно. Это, к сожалению, не относится к цифровой информации, потому что, если мы в файле заменим один бит на другой, никто этого не заметит, а использовать обычную подпись нельзя, так как есть фотошоп, в котором можно ее подделать. Таким образом, нам нужны исключительно математические методы, которые обеспечат невозможность отказа отправителя и получателя от факта передачи сообщения, это назвали *цифровой подписью*.

Криптографические механизмы безопасности

Перечислим основные механизмы, с помощью которых мы будем обеспечивать работу сервисов безопасности:

- *Алгоритмы симметричного шифрования*. На вход алгоритма подается исходное сообщение и ключ, а выходом является шифрованное сообщение. Сам алгоритм всем известен, но без знания ключа невозможно получить исходное сообщение. Соответственно, существует механизм расшифрования, на вход которого

подается шифрованный текст и тот же самый ключ (отсюда название симметричное шифрование), на выходе имеем исходное сообщение. Такой алгоритм называют криптографически стойким: даже если известно много пар исходного и шифрованного текста, полученных с использованием одного и того же ключа, нельзя найти ключ, который использовался для шифрования. Конечно, можно перебирать все множество ключей, так называемая, атака грубой силы, чтобы защититься от этого, нужно использовать алгоритмы с большой длиной ключа.

- *Алгоритмы асимметричного шифрования*. В этом случае используется два ключа: открытый (KU) и закрытый (KR). Особенность таких алгоритмов заключается в том, что зная открытый ключ, вычислительно невозможно получить соответствующий ему закрытый. Это свойство как раз позволит нам создавать сервис невозможности отказа или цифровую подпись.

- *Хэш-функции*. Они позволяют создавать из сообщения произвольной длины маленький хэш-код этого сообщения (его «отпечатки пальцев»), по которому можно восстановить исходное сообщение. Если в исходном сообщении изменится один бит, то в хэш-коде изменится уже достаточно большое количество битов. Однако одному хэш-коду могут соответствовать несколько сообщений, об этом мы подробнее поговорим, когда будем рассматривать хэш-функции.

Модель сетевой безопасности

Итак, у нас есть отправитель и получатель, и мы хотим создать безопасный информационный канал, который защищал бы нас от нарушителя. Во многих случаях нам потребуется третья доверенная сторона TTP (Third Trust Party), к которой и отправитель и получатель должны будут получить доступ либо заранее, либо непосредственно перед установлением соединения. Помощь TTP будет каждый раз разная, мы будем говорить о ней в каждом конкретном случае. И отправитель и получатель должны «доверять» TTP (рис. 1.10).

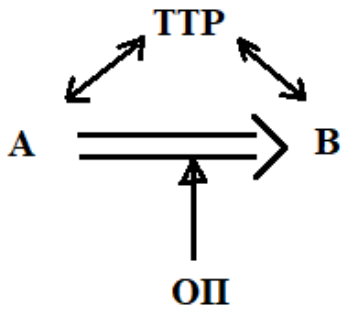

Рис. 1.10. Модель сетевой безопасности.

Модель информационной системы

В качестве информационной системы может предстать и один компьютер, и сеть, главное, чтобы у нее были точно определенный периметр и точки входа, и тогда про любой компьютер можно сказать: внутри он этой системы или вне ее. Для обеспечения безопасности этой системы нам понадобится, во-первых, чтобы в каждой точке входа стояла некая граничная функция, которую мы будем называть *межсетевым экраном*, он будет разрешать или запрещать прохождение трафика через точку входа. Во-вторых, должны быть обеспечены средства разграничения доступа, то есть если какой-то трафик прошел внутрь системы, это не означает, что ему должно быть доступно все. В-третьих, нам необходимы системы обнаружения проникновения IDS (Intrusion Detection System) или появившиеся сейчас системы предотвращения проникновения IPS (Intrusion Prevention System). Если произошло нарушение политики безопасности, это не должно остаться незамеченным.

ЛЕКЦИЯ 2. АЛГОРИТМЫ СИММЕТРИЧНОГО ШИФРОВАНИЯ. DES, BLOWFISH

Алгоритмы симметричного шифрования

Алгоритм мы будем обозначать буквой E (Encrypt), на вход алгоритма подается исходное сообщение P (Plain text) и ключ K (Key), выходом является зашифрованное сообщение C (Cipher text). Существует также алгоритм расшифрования D (Decrypt), на вход которого подается зашифрованное сообщение C и тот же самый ключ K, а выходом является исходное сообщение P (рис. 2.1).

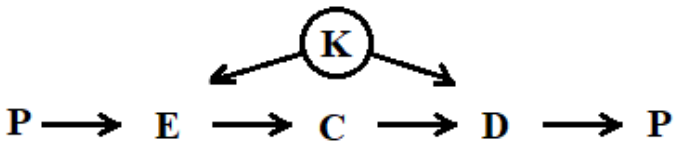

Рис. 2.1. Схема алгоритма симметричного шифрования.

Для шифрования и расшифрования используется один и тот же ключ, или же ключ расшифрования можно легко получить из ключа шифрования.

Первое что мы хотим от алгоритмов симметричного шифрования – это конфиденциальность. Цель нарушителя – узнать исходное сообщение или ключ и по нему расшифровать сообщение. Вводится понятие *криптостойкости,* алгоритм называется *криптостойким*, если по зашифрованному сообщению нельзя узнать исходное сообщение и узнать ключ никаким путем, кроме как простым перебором всего пространства ключей, так называемой, brute force attack. Если длина ключа n бит, то имеем 2^n возможных вариантов ключей. Нам никуда не деться от этой атаки, она всегда имеет место, единственный вариант защиты – использование большей длины ключа. Таким образом, весь секрет алгоритмов симметричного шифрования заключается в ключе, встает вопрос, как передать ключ от отправителя к получателю, чтобы нарушитель не мог его узнать.

Введем несколько новых понятий. *Диффузия* – это статистическая зависимость между исходным и зашифрованным сообщением. Хорошей диффузии (хороший лавинный эффект) соответствует ситуация, когда изменение одного бита исходного сообщения приводит к изменению достаточно большого количества бит в зашифрованном сообщении, или на значение одного бита в зашифрованном сообщении влияет достаточно большое количество бит исходного сообщения.

Конфузия – это статистическая зависимость между ключом и зашифрованным сообщением. И снова изменение одного бита ключа должно приводить к изменению достаточно большого количества бит в зашифрованном сообщении.

Рассмотрим примеры алгоритмов симметричного шифрования. Первый алгоритм, который известен человечеству – это «шифр Цезаря». В нем каждая буква заменяется на +k, например, a на d, b на e и так далее, таким образом k является ключом. Этот алгоритм не является криптостойким, потому что в любом языке частота появления каждой буквы известна, например, мы знаем, что буква a встречается намного чаще, чем z, таким образом, мы можем определить буквы по частоте их появления, даже не ища ключ.

Все современные алгоритмы делятся на два класса: *блочные* и *поточные. Поточные* алгоритмы – это алгоритмы почти столетней давности, которые на сегодняшний день почти не применяются. *Блочный* алгоритм разбивает сообщение на блоки определенной длины, и шифруется отдельный блок.

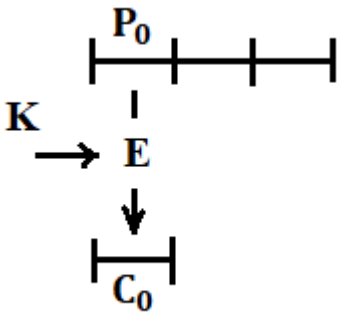

Рис. 2.2. Схема блочного алгоритма.

Блок исходного сообщения последовательно пропускается через одну и ту же функцию, но с разными ключами, последовательность преобразований называется *раундами*. Выход предыдущего раунда является входом для следующего. Количество раундов r является параметром алгоритма, а также длина ключа и длина блока. Ключи раундов K_i получаются из исходного ключа K по-своему для каждого алгоритма.

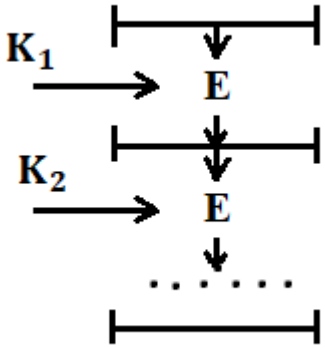

Рис. 2.3. Схема шифрования отдельного блока.

Описать алгоритм симметричного шифрования – значит задать функцию раунда, одинаковую для каждого раунда. В некоторых алгоритмах первый и последний раунды немного отличаются, в первых блочных алгоритмах первые и последние преобразования, так называемые, забеливания (whitening) вообще выполнялись без использования ключа и никакой дополнительной безопасности не давали. В современных алгоритмах первые и последние преобразования могут отличаться, но всегда выполняются с использованием ключа.

Поговорим о том, какие операции могут содержать функции раундов.

1. XOR (исключающее «или» – $\oplus$) является наиболее часто используемой операцией, вы не найдете алгоритма симметричного шифрования, в котором она отсутствует.
2. S-боксы (S-box): i бит входного сообщения преобразуется в j бит выходного сообщения. Говорят, что размерность S-box равна $i \times j$. Перемешивание текста в основном выполняется в S-боксах.
3. Операции сложения по модулю 2^{32}. Степень имеет значение 32, поскольку до сих пор предполагается, что самые распространенные устройства, которым необходима конфиденциальность, и которые, соответственно, должны использовать алгоритмы симметричного шифрования, имеют 32-битную архитектуру.
4. Операция сложения по модулю 2^{16}.
5. Циклические сдвиги на L бит ($\ll L$).
6. Перестановки всего блока (P).

Существуют требования к алгоритмам симметричного шифрования.

1. Безопасность. Оно означает, что не должно быть известных атак. Мы часто будем говорить фразу: «атака на сегодняшний день не найдена».
2. Эффективность. Грубо говоря, безопасность сама по себе не является тем, ради чего мы покупаем компьютеры.
3. Дополнительным требованием можно указать плоское пространство ключей: любую последовательность бит нужной длины можно использовать в качестве ключа шифрования и данный ключ будет стойким, то есть, зная только зашифрованное сообщение, нельзя найти ни ключ, ни исходное сообщение. Это достаточно жесткое требование, как правило, чаще слабые ключи существуют, их немного, и они заранее известны.
4. Простота. Алгоритм должен быть простым, чтобы его можно было проанализировать на поиск атак, слабых ключей.
5. Модифицируемость. Алгоритм должен быть легко модифицируемым, для того чтобы использоваться в случае, когда сильнее требования к защите и когда сильнее требования к эффективности.

Сеть Фейштеля

Рассмотрим как может быть реализована функция E (Encrypt). В основе многих алгоритмов лежит, так называемая, сеть Фейштеля (Feistel). Блок делится на левую и правую часть, за один раунд преобразуется только половина блока, и выполняется ее XOR с другой половиной блока. Результат помещается в левую часть выходного блока, а левая часть исходного блока без изменения помещается в правую часть выходного блока (рис. 2.4): $L_i = R_{i-1} \oplus F(L_{i-1}, K_i)$ и $R_i = L_{i-1}$.

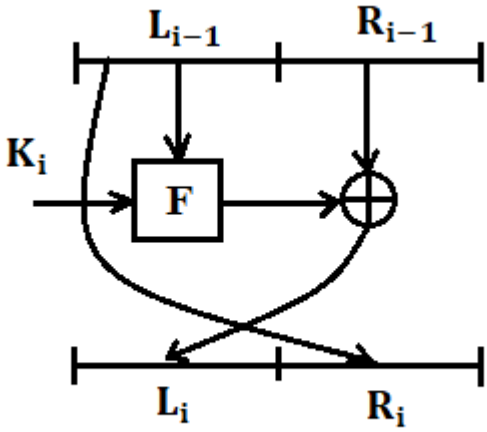

Рис. 2.4. Схема сети Фейштеля.

Это, так называемая, сеть Фейштеля с двумя ветвями, то есть за один раунд у нас изменяется только половина блока.

Сеть Фейштеля хороша тем, что для расшифрования сообщения достаточно просто подавать ключи в обратном порядке, нам не требуется обратимости функции F. Таким образом, задача функции F заключается в том, чтобы хорошо перемешать левую половину блока, а расшифровать ее мы всегда сможем.

Алгоритм DES

Первым алгоритмом, который мы рассмотрим, будет алгоритм DES (Date Encryption Standard). Он был разработан в 1977 году и принят NIST-ом (National Institute of Standards and Technology) в качестве стандарта алгоритмов симметричного шифрования. Длина ключа составляет 56 бит, соответственно всего ключей 2^{56}, на сегодняшний день это не так уж и много. В 1995-1996 годах был произведен конкурс на взлом DES путем атаки грубой силы, и уже в те годы алгоритм был взломан за три часа. Однако это лучше, чем передавать данные вообще в открытом виде. На самом деле, алгоритм DES много где реализован, поскольку он «вшит в железо», а нам незачем выбрасывать купленное «железо», если оно работает.

Другие параметры алгоритма DES: длина блока - 64 бита. В прошлом веке меньшую длину блока было иметь нельзя, потому что никто не стал бы искать ключ, все бы просто перебирали блоки. Сейчас мощности компьютеров возросли, и длина блока не бывает меньше 128 бит. Количество раундов составляет 16. Алгоритм использует классическую сеть Фейштеля. Существуют начальная и конечная перестановки, которые задаются таблицей IP (Initial Permission). Начальная перестановка выполняется без использования ключа, поскольку в 1977 году не понимали, что такая перестановка не несет никакой дополнительной безопасности. Далее выполняются 16 раундов алгоритма, меняются местами левая и правая части, и наконец выполняется перестановка IP^{-1}, обратная начальной.

Рассмотрим отдельный раунд алгоритма (рис. 2.5). Во-первых, мы разделяем блок на левую и правую части по 32 бита. Затем происходит расширение правой ветки до 48 бит, обозначим ее (abcdifgij…), где буквы – это биты, принимающие значения 0 и 1. Ветка разбивается на группы по 4 бита, и к каждой группе справа и слева циклически добавляем по одному биту. Таким образом, имеем группы по 6 бит, и из 32-битной ветки получаем 48-битную. Далее выполняется XOR нашей новой ветки с 48-битным ключом раунда. Результат пропускается через 8 S-боксов размерностью 6×4, ветка делится на группы по 6 бит, средние 4 бита определяют номер столбца, а два крайних

бита – номер строки, в соответствии с которыми мы находим в таблице S-бокса выходное значение. Основное внимание в разработке DES уделялось именно разработке S-боксов. Главная задача заключалась в том, чтобы, оставляя S-боксы в открытом доступе (это позволяет производить анализ слабых мест алгоритма, создавать open source реализации, дешевые аппаратные реализации), максимально скрыть статистические закономерности исходного текста. Вернемся к описанию алгоритма, далее выполняется перестановка всей ветки таким образом, чтобы в следующем раунде каждый бит обрабатывался другим S-боксом, и в конце выполняется XOR всего этого с левой частью (рис. 2.5). Итог помещается в правую часть: $R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$, а правая часть без изменения помещается в левую: $L_i = R_{i-1}$.

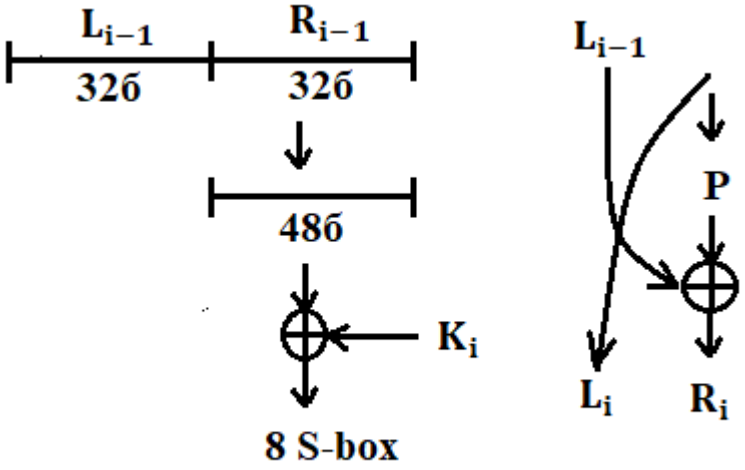

Рис. 2.5. Отдельный раунд алгоритма DES.

Давайте покажем, что для расшифрования сообщения нам не требуется обратимость функции F. Верхним индексом d будем обозначать процедуру расшифрования. На вход алгоритма расшифрования подается зашифросанный текст C, пропущенный через начальную перестановку: $L_0^d R_0^d = IP(C)$. Зашифрованный текст с точки зрения шифрования есть перестановка, обратная начальной: $C = IP^{-1}(L_{16}, R_{16})$. Поскольку перестановки IP и IP^{-1} взаимоисключающие, получим: $L_0^d R_0^d = L_{16} R_{16}$, то есть на вход первого раунда алгоритма расшифрования подается выход последнего раунда алгоритма шифрования. Теперь нам надо показать, что выходом первого раунда алгоритма расшифрования является вход последнего раунда алгоритма шифрования. Имеем следующие выражения:

$$L_1^d = R_0^d = L_{16} = R_{15} ,$$

$$R_1^d = L_0^d \oplus F\left(R_0^d, K_{16}\right) = R_{16} \oplus F(L_{16}, K_{16}) = L_{15} \oplus F(R_{15}, K_{16}) \oplus F(L_{16}, K_{16}) = L_{15}.$$

Таким образом, мы доказали возможность расшифрования сообщения.

Итак, зачем использовать DES, если его можно довольно быстро взломать. Во-первых, когда мы выходим в интернет, не все наши коммуникации защищены, и лучше использовать DES, чем ничего. Во-вторых, он достаточно эффективный, не требует много времени. В-третьих, не всегда мы делаем в интернете что-то такое, что нас захотят взламывать. В-четвертых, DES везде уже по умолчанию реализован.

Тем не менее, в случае, когда конфиденциальность информации нам важна, необходим более надежный алгоритм. Разрабатывать новый долго, поэтому возникает идея использовать один и тот же алгоритм несколько раз.

Двойной DES

У нас есть исходное сообщение, мы шифруем его одним ключом $E_{K_1}[P]$, а потом зашифрованное сообщение шифруем еще раз другим ключом $C = E_{K_2}\left[E_{K_1}[P]\right]$. Длина каждого ключа составляет 56 бит, хочется получить стойкость алгоритма $2^{2\cdot 56}$, но мы ее не получим. Предположим, что нарушитель знает одну пару $(P_0C_0)_{K_1,K_2}$, полученную с использованием нашей пары ключей, а это вполне реально. Он шифрует на всех ключах известный ему текст и получает 2^{56} значений: $X = E_K[P_0]$, далее он расшифровывает на всех ключах известный ему шифрованный текст и получает еще 2^{56} значений: $Y = D_K[C_0]$. И наконец сравнивает, какие значения совпали. В итоге он потратил всего 2^{57} усилий, что не намного больше усилий, затрачиваемых на взлом одного DES алгоритма. Мы не получили желаемую стойкость $2^{2\cdot 56}$, а только сами потратили больше усилий, поэтому алгоритм двойного DES непригоден.

Тройной DES

Рассмотрим алгоритм тройного DES, в этом случае мы исходное сообщение P шифруем три раза подряд: $C = E_{K_3}\left[E_{K_2}\left[E_{K_1}[P]\right]\right]$. Стоит помнить, что мы еще должны передать $3 \cdot 56$-битный ключ получателю, чтобы он имел возможность расшифровать наше сообщение. Мы увидим, что сложность алгоритмов, которые умеют безопасно передавать значение ключа на противоположную сторону, чтобы нарушитель не мог его перехватить, возрастает экспоненциально с увеличением длины ключа. Вообще говоря, идеальный шифратор это XOR с абсолютно случайным ключом той же длины, что и исходное сообщение, но ведь мы не решаем задачу, это одно и то же, что передать исходное сообщение безопасно, что ключ. Поэтому до недавнего времени использовался ключ $K_3 = K_1$, мы получаем тройной DES с двумя ключами. Если в

смысле эффективности нас устраивает длина 112 бит, то вполне можем использовать этот алгоритм. Часто алгоритм используется в, так называемом, режиме EDE: $C = E_{K_1}\left[D_{K_2}\left[E_{K_1}[P]\right]\right]$.

Алгоритм Blowfish

Алгоритм Blowfish разработан Брюсом Шнайером и интересен тем, что он очень хорошо подходит для сетевого взаимодействия. Как происходит установление соединения, грубо говоря, есть этап «рукопожатия», то есть согласования каких-то параметров соединения, далее довольно быстро начинается обмен данными. Алгоритм Blowfish также имеет большую стадию начальной выработки параметров, а в остальном он очень быстрый. Параметры алгоритма: длина ключа не фиксирована и может быть достаточно большой до 448 бит, длина блока – 64 бита, количество раундов – 16. Заметим, что это сеть Фейштеля, то есть

$$L_i = R_{i-1} \oplus F(L_{i-1}),$$

$$R_i = L_{i-1} \oplus K_i.$$

Разберемся, что собой представляет функция F. У нас есть 32 бита L_{i-1}, мы разбиваем их на 4 части по 8 бит и обозначаем A, B, C и D. Далее имеем 4 S-бокса S_{1A}, S_{2B}, S_{3C} и S_{4D}, на вход которых подается 8 бит, а выходом является 32-битное значение. Искомая функция имеет вид:

$$F = \left(\left((S_{1A} + S_{2B}) mod 2^{32} \oplus S_{3C}\right) + S_{4D}\right) mod 2^{32}.$$

Особенность в том, что S-боксы зависят от ключа, например, в DES они были фиксированы. За это алгоритм подвергался критике, так как не выполнено условие простоты.

Как создаются эти S-боксы? У нас есть их начальные стандартные значения, далее выполняется расширение ключа с помощью конкатенации на нужного размера. Затем шифруется стандартная строка P_0 с использованием стандартных S-боксов и нашего ключа, а результат заменяет ключи K_1, K_2. Этот результат опять шифруется уже с новыми ключами, в итоге ключи заменяются на K_3, K_4. Таким образом, каждый раз меняются два ключа. Затем мы начнем менять выходы S-боксов, так же по два за одно шифрование.

ЛЕКЦИЯ 3. АЛГОРИТМЫ СИММЕТРИЧНОГО ШИФРОВАНИЯ. AES, RIJNDAEL

Алгоритм ГОСТ 28147

ГОСТ 28147 – это алгоритм симметричного шифрования, разработанный в СССР, он был принят в качестве стандарта в 1989 году. Тогда к алгоритмам относились как к секретной информации, поэтому, несмотря на существование международных стандартов, считали необходимым иметь свой алгоритм, принцип работы которого не раскрывался. ГОСТ 28147 является классической сетью Фейштеля, длина блока – 64 бита, длина ключа – 256 бит, количество раундов – 32, также имеют место соотношения:

$$L_i = R_{i-1},$$

$$R_i = L_{i-1} \oplus F(R_{i-1}).$$

Функция F имеет очень простой вид: сложение с ключом K_i по $mod 2^{32}$, 8 S-боксов 4×4 и циклический сдвиг на 11 бит влево $\ll_{11}$. Стоит отметить, что помимо того, что S-боксы нельзя публиковать, их следует достаточно часто менять, но, конечно, не так часто как ключ. В DES же S-боксы не меняют, поскольку этот алгоритм есть везде, и замену произвести затруднительно, а вот ГОСТ 28147 используется в узкоспециализированных государственных структурах.

Алгоритм IDEA

Алгоритм симметричного шифрования IDEA (International Data Encryption Algorithm) разработан в Швейцарии в 1991-1992 году. Интересно то, что долгое время данный алгоритм рассматривался как замена DES, поскольку уже в 90-е годы считали, что длина ключа 56 бит маловата. Алгоритм IDEA использовался в PGP (программа, обеспечивающая безопасность электронной почты), по крайней мере, в первых ее версиях (PGP была популярна в 90-е годы, сейчас активно используется SSL). Отметим, что алгоритм IDEA не является сетью Фейштеля. Параметры IDEA: длина ключа – 128 бит, длина блока – 64 бита, количество раундов – 8 и, так называемое заключительное преобразование. IDEA содержит операции не характерные для алгоритмов симметричного шифрования: XOR, сложение по $mod\ 2^{16}$, умножение по $mod(2^{16}+1)$. Эти операции не совместимы между собой, то есть для любой пары из трех операций не действует дистрибутивный закон и не действует ассоциативный закон.

В основе работы алгоритма лежит блок MA (Multiply Addition): на вход подается два 16-битных значения P_1, P_2 , используется сложение, умножение с ключами K_5 и K_6 , на выходе получаем C_1, C_2 (рис. 3.1).

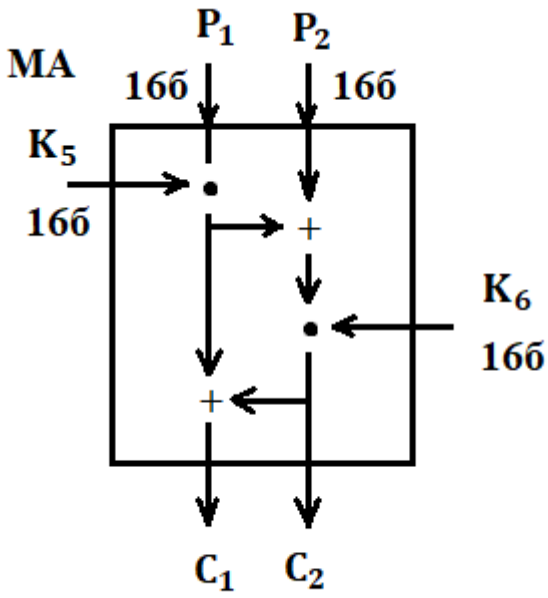

Рис. 3.1. Схема MA блока.

Исходное сообщение делится на 4 части по 16 бит: P_1, P_2, P_3, P_4. Далее к каждому применяются соответствующие операции сложения и умножения, и блок MA, на выходе вновь имеем четыре 16-битных значения уже зашифрованного текста (рис. 3.2). Алгоритм содержит 8 таких раундов и одно заключительное преобразование.

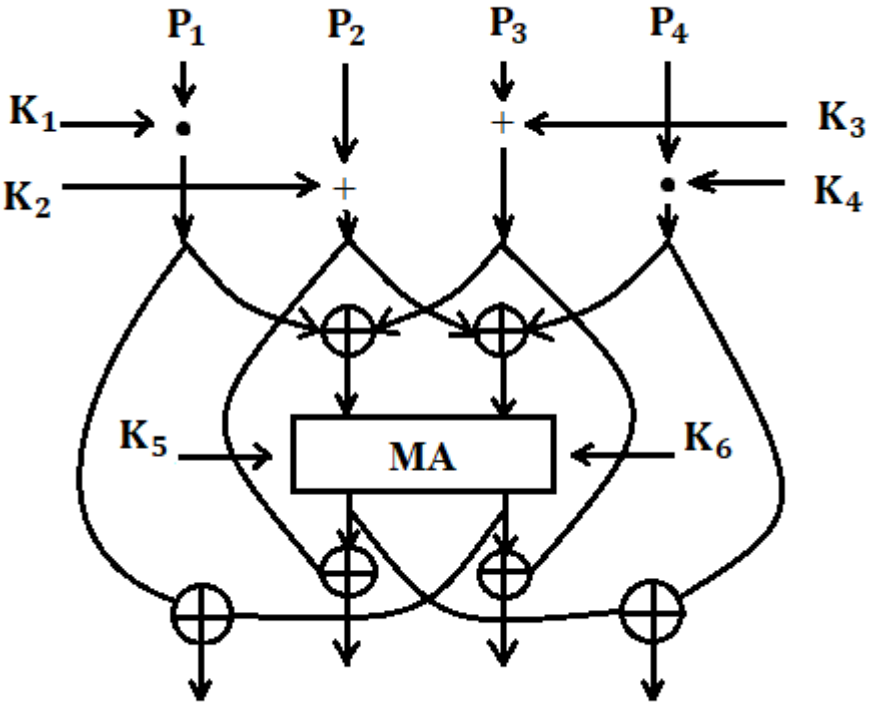

Рис. 3.2. Схема раунда алгоритма IDEA.

Нам нужно всего $(8 \cdot 6 + 4)$ 16-битных подключей, соответственно, $(8 \cdot 6 + 4) \cdot 16$ бит ключевого материала. Исходный ключ имеет длину 128 бит, первые 4 ключа нарезаются из 128-битного ключа, затем выполняется циклический сдвиг на 25 бит $\ll_{25}$, чтобы первый ключ в каждом раунде брался из своей области исходного ключа, и нарезаются новые ключи по 16 бит. Алгоритм достаточно быстрый, поскольку содержит небольшое количество раундов.

Алгоритм AES

В 2001 году NIST принял новый стандарт AES (Advanced Encryption Standard). К новому алгоритму на замену DES выдвигали следующие требования: алгоритм должен быть симметричным, блочным, итерационным, длина ключа должна принимать значения 128, 192 или 256, длина блока должна быть не меньше 128 бит. Это говорит о том, что мощности компьютеров к 2000 году значительно возросли. В результате упомянутых требований алгоритм будет работать медленнее, но это необходимо для защиты от атаки грубой силы. Не будем забывать и о требовании безопасности, на

алгоритм не должно быть известных атак. Алгоритм должен быть эффективен в различных окружениях: на маленьких устройствах с маленькой оперативной и внешней памятью и на больших компьютерах. Если алгоритм будет основан на сети Фейштеля, а длина блока, напомним, составляет 128 бит, то у нас будет четыре ветки вместо двух, и, соответственно, за один раунд будут изменяться две ветки. Мы уже говорили о том, что алгоритм должен быть открытым. Он также не должен находиться ни в чьей собственности: ни государства, ни какой-либо частной компании, и не должен иметь никаких патентных отчислений. Но все-таки понятие стоимости алгоритма существует, под этим понимается используемая оперативная, внешняя память, процессор.

В итоге к 2000 году было отобрано 5 алгоритмов:

- MARS – сеть Фейштеля с четырьмя ветвями (IBM)
- RC6 – сеть Фейштеля (компания RSA)
- Rijndal – не сеть Фейштеля, принят в качестве AES, поскольку показал одинаковую эффективность как на маленьких, так и на больших компьютерах.
- Serpent – не сеть Фейштеля
- Twofish – сеть Фейштеля

Алгоритм Rijndal

Rijndal – это, так называемый, байт-ориентированный алгоритм. Представим блок следующим образом:

$$A_{00}A_{10}A_{20}A_{30}A_{01}A_{11}A_{21}\dots,$$

он разбит на части по 8 бит, далее перепишем его в виде:

$$\begin{matrix} A_{00} & A_{01\dots} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \\ A_{30} & A_{31} \end{matrix}.$$

Такую запись блока будем называть *состоянием* алгоритма. Длина блока может принимать значения 128, 192, 256 бит. Один столбец занимает 32 бита, то есть мы имеем 4, 6 или 8 столбцов в зависимости от длины блока в битах ($N_B = 4,6,8$). Аналогичные рассуждения применяются и к ключу: длина ключа может принимать значения 128, 192, 256 бит или 4, 6, 8 столбцов ($N_K = 4,6,8$). Количество раундов N_R зависит от комбинации значений длины ключа и длины блока (табл. 3.1). Отметим, что современные алгоритмы имеют больше параметров, например, мы уже не обязаны фиксировать количество раундов строго на 16-ти. Увеличение количества раундов

повышает стойкость алгоритма. Это связано с тем, что поиск атаки на алгоритм симметричного шифрования ведется следующим образом: ищется атака на алгоритм с меньшим количеством раундов, а если найдена атака на алгоритм с n раундами, то, как правило, для алгоритма с n+1 раундами она пропадает. Но, к сожалению, увеличивается время выполнения алгоритма, то есть снижается его эффективность.

N_R	$N_B = 4$	6	8
$N_K = 4$	10	12	14
6	12	12	14
8	14	14	14

Таблица 3.1. Количество раундов N_R в зависимости от длины блока N_B и длины ключа N_K.

Рассмотрим один раунд алгоритма Rijndal, он зависит от состояния и ключа раунда и состоит из четырех слоев: подстановка, сдвиг строк, перемешивание столбцов, добавление ключа раунда.

Round(State, RoundKey)={ByteSub(State), ShiftRow(State), MixColumn(State), AddRoundKey(State, RoundKey)}.

Помним, что это не сеть Фейштеля, поэтому каждое преобразование должно быть обратимо. Разберем каждый слой:

1. ByteSub.

   Пусть у нас есть состояние $\begin{matrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & \\ A_{30} & \end{matrix}$, преобразуем его в состояние $\begin{matrix} B_{00} & B_{01} \\ B_{10} & B_{11} \\ B_{20} & \\ B_{30} & \end{matrix}$.

   Каждый байт остается на своем месте и с помощью S-бокса преобразуется в другой байт. В алгоритмах, которые мы рассматривали ранее, нам нигде не требовалась обратимость S-боксов. С одной стороны, нам нужна хорошая диффузия, а с другой стороны, нужна обратимость. Здесь уже привлекается математика, давайте каждый байт представим в виде полинома 7-й степени:

   $$a(x) = a_7x^7 + \cdots + a_1x + a_0, a_i = \{0,1\}.$$

   Мы хотим для этих полиномов определить поле. Если мы определим поле над этим конечным множеством полиномов, это будет означать, что у нас есть единственный обратный элемент. Для этого нам нужно определить операции сложения и умножения. Операция сложения – это просто XOR. Операция умножения полиномов $a(x)$ и $b(x)$ определяется несколько сложнее: как остаток от деления обычного, известного нам из школы, произведения $a(x)b(x)$ на полином $m(x)$ 8-й степени, примем его равным

$$m(x) = x^8 + x^4 + x^3 + x + 1,$$

это простой полином, то есть его нельзя разложить на сомножители. Обратный элемент определяется из соотношения

$$a(x)b(x) = 1 \left(mod\big(m(x)\big)\right).$$

2. ShiftRow.

Если есть строка $a_{00}a_{01}a_{02}a_{03}$, то сдвинуть ее на одну позицию означает получить строку $a_{01}a_{02}a_{03}a_{00}$. У нас

$C_0 = 0$ (нулевая строка у нас остается без изменения)

$C_1 = 2$,

$C_2 = 2$ ($N_B = 4,6$) и $C_2 = 3$ ($N_B = 8$),

$C_3 = 3$ ($N_B = 4,6$) и $C_3 = 4$ ($N_B = 8$).

3. MixColumn.

Каждое состояние $\begin{matrix} A_{00} & A_{01} \\ A_{10} & \\ A_{20} & \\ A_{30} & \end{matrix}$ рассматривается как полином 3-й степени $A_3x^3 + A_2x^2 + A_1x + A_0$, где коэффициенты являются байтами $A_i \in \{2^8\}$. Наша задача заключается в том, чтобы перемешать значения коэффициентов обратимым образом. Выбираются уже полиномы $M(x) = x^4 + 1$ и $c(x) =' 03'x^3 +'01'x^2 +'01'x +'02'$, они являются взаимно-простыми, тогда

$$b(x) = a(x)c(x) \left(mod\big(M(x)\big)\right).$$

Соответственно, у этого полинома существует обратный полином

$$d(x) =' 0B'x^3 +'0D'x^2 +'09'x +'0E'.$$

Плюсы Rijndal. На сегодняшний день атак этого алгоритма не найдено, если будет найдена атака, то в первую очередь можно будет изменить полиномы $M(x)$ и $m(x)$, но не будем забывать, что это большая работа, требующая переписывания всего программного обеспечения. Алгоритм легко масштабируется с шагом 32, можно легко расширять длину блока, длину ключа, если это необходимо.

Минусы Rijndal. Никто не утверждал, что представление байта в виде полинома является единственно возможным представлением. Если будет найдено другое представление байта, то может быть найдена атака.

Режимы выполнения алгоритмов симметричного шифрования

В действительности нам необходимо шифровать и очень большие сообщения, если речь идет о HTML странице или файле в Word, и очень маленькие, для этого существуют особые режимы выполнения алгоритмов симметричного шифрования:

1. ECB (Electron Code Book).
   Сообщение разбивается на блоки требуемой длины, которая равна длине блока алгоритма симметричного шифрования. Последний блок добавляется так, чтобы он был кратным блоку алгоритма симметричного шифрования. Все блоки независимо один от другого шифруются с одним и тем же ключом. Возникает важное обстоятельство: если у нас по какой-то причине два исходных блока одинаковые, то и соответствующие зашифрованные блоки будут одинаковые, а это дает дополнительную информацию нарушителю.

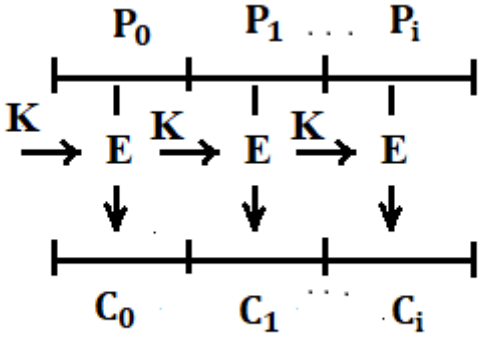

Рис. 3.3. Схема ECB режима.

2. CBC (Cipher Block Chaining) – цепочка зашифрованных блоков.
   В этом случае сообщение так же разбивается на блоки требуемой длины, последний блок добавляется. Дополнительно к ключу вводится еще один параметр IV, который называется *инициализационный вектор*. Выполняется XOR инициализационного вектора с первым блоком исходного сообщения, результат операции подается в алгоритм шифрования. Далее выполняется XOR предыдущего зашифрованного блока с текущим незашифрованным блоком, и результат подается в алгоритм шифрования. Таким образом, решается проблема, существующая в режиме ECB: если два блока исходного сообщения одинаковые, то соответствующие зашифрованные блоки будут уже разные.

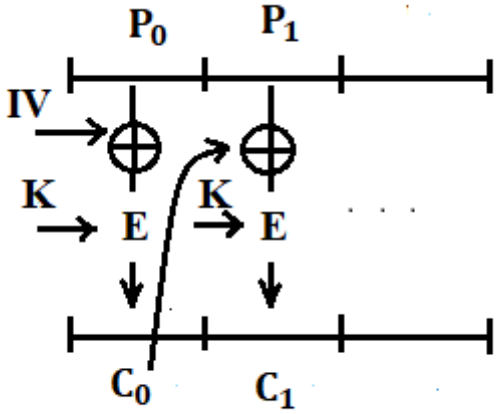

Рис. 3.4. Схема CBC режима.

Но теперь получатель помимо ключа должен знать еще и инициализационный вектор, это может быть проблемой, поскольку его тоже нужно безопасно передать. Поэтому если требования к безопасности не очень строгие, тогда делаем инициализационный вектор равным нулю. Иначе задаем его случайным образом так же, как и ключ. Этот режим хорош для шифрования больших сообщений: HTML-страница, файл в Word.

ЛЕКЦИЯ 4. КРИПТОГРАФИЯ С ОТКРЫТЫМ КЛЮЧОМ. АЛГОРИТМ RSA

Режимы выполнения алгоритмов симметричного шифрования

Продолжим рассматривать классификацию режимов выполнения алгоритмов симметричного шифрования, которую мы начали в прошлой лекции:

1. ECB (Electron Code Book)

2. CBC (Cipher Block Chaining)

3. CFB (Cipher Feedback) – шифратор с обратной связью.
   По факту это поточный шифратор. Исходный блок называется регистром сдвига, он подается в алгоритм шифрования, далее берутся левые J бит, и выполняется XOR с незашифрованными J битами P_0, в итоге получается J бит зашифрованного текста C_0. Затем выполняется сдвиг на J бит $<_J$ регистра сдвига, и значение C_0 помещается в правые J бит регистра сдвига. И так далее. Число бит J может быть любым, но, разумеется, меньше длины блока, в большинстве случаев это 1 байт. Итак, мы получили поточный шифратор, теперь приложению не надо ждать, когда у него наберется 64, 128, 256 бит сообщения. Недостаток этого режима заключается в том, что возможен ненадежный транспорт битов от отправителя к получателю, например, UDP не гарантирует надежный транспорт пакетов. Если у нас потерялся один пакет, то потерялся весь оставшийся хвост. Это было быне смертельно, если бы у нас не было ничего зашифровано, например, для IP-телефонии, видеоконференций это не критично.

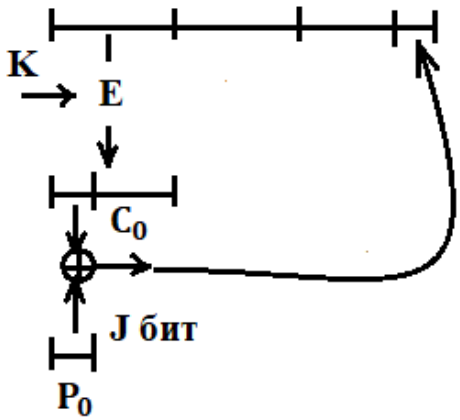

Рис. 4.1. Схема CFB режима.

4. OFB (Output Feedback) – шифрование по выходу.
   У нас снова есть регистр сдвига, который мы шифруем с использованием ключа. Выделяем левые J бит, выполняем XOR с J битами исходного текста и получаем C_0, которое отправляем получателю. Далее так же осуществляем сдвиг на J бит влево $<_J$, но теперь в конец регистра сдвига добавляем J бит до выполнения XOR. Таким образом, в регистре сдвига постоянно крутится зашифрованное исходное значение. Регистр сдвига не зависит от P (plain text). Преимущество данного режима заключается в том, что в случае потери пакета при транспорте от отправителя к получателю ничего страшного не произойдет – следующий пакет расшифруется. Однако мы теряем в безопасности, потому что в хвост регистра мы фактически помещаем зашифрованный исходный текст, и можно попытаться перебрать ключи.

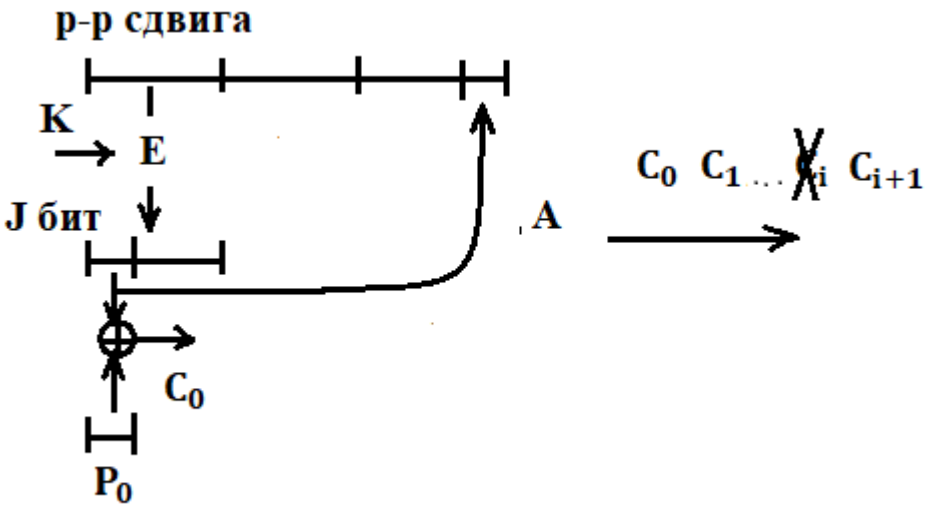

Рис. 4.2. Схема OFB режима.

Генераторы псевдослучайных чисел

Для чего нам нужны случайные числа?

1. Ключ шифрования.
   Случайные числа нужны нам в первую очередь для написания ключа. Если ключ будет создаваться каким-то заранее известным алгоритмом $K = f(K)$, то

нарушитель всегда сможет найти его, зная одно из предыдущих значений, и все наши разговоры о крипто-устойчивости становятся бессмысленными.

2. Nonce (Number only once) – число, которое используется только один раз.
   На первой лекции мы рассматривали relpay-атаки: отправитель A передает получателю B некий аутентификатор в защищенном виде, его захватывает нарушитель и через какое-то время Δt передает его получателю. B должен понять, что это повторное использование одного и того же аутентификатора. В каждый такой аутентификатор помещается число nounce, которое будет использоваться только один раз, то есть оно должно быть случайным.

Какие числа мы будем называть случайными или псевдослучайными? Имеем следующие требования:

1. Равномерное распределение.
2. Непредсказуемость.
   Если нарушитель знает n чисел $r_1 \dots r_n$, созданных генератором псевдослучайных чисел, он не должен иметь возможность вычислить следующее случайное число r_{n+1}.

Поговорим об источниках случайных и псевдослучайных чисел. Случайные числа создаются физическими приборами, а псевдослучайные – математическими функциями. Понятно, что для наших целей создавать реальный физический шум будет затруднительно. Алгоритмы симметричного шифрования помимо обеспечения конфиденциальности используются как генераторы псевдослучайных чисел. Простейшая схема такого генератора представлена на рис. 4.3. Нам необходимо засекретить только ключ.

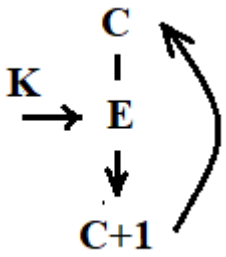

Рис. 4.3. Простейшая схема генераторы псевдослучайных чисел на основе алгоритма симметричного шифрования

Стандартом для генерации псевдослучайных чисел является ANSI X 9.17, в котором трижды используется тройной DES с двумя ключами в режиме EDE (Encrypt Decrypt Encrypt). На вход алгоритма подаются текущие дата и время DT_i, и два ключа

K_1, K_2, производится XOR результата первого DES с текущим значением вектора V_i, результат операции снова отправляется на вход следующего тройного DES с теми же ключами. Выход и будет текущим случайным числом R_i. Для получения нового вектора V_{i+1}, который будет использоваться при следующем вызове генератора, производится XOR R_i с выходом первого тройного DES, а результат подается в третий тройной DES (рис. 4.4).

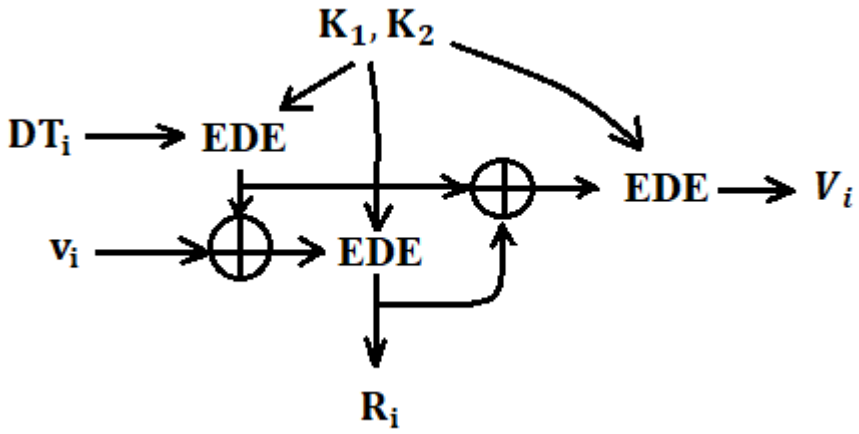

Рис. 4.4. Схема генератора псевдослучайных чисел ANSI X 9.17

Получаемое нами случайное число R_i зависит от ключей, текущих значений даты и времени (это не секретный компонент, однако он каждый раз разный) и предыдущего значения вектора (должно быть секретным, как и ключ). Безопасно хранить необходимо ключи и текущее значение вектора.

Алгоритмы асимметричного шифрования. Криптография с открытым ключом

Симметричная криптография не решает следующих проблем. Во-первых, если участники ничего не знают друг о друге, то есть у них нет никакого общего секрета, то они не смогут установить защищенное соединение. Во-вторых, нам необходим некий аналог подписи, отправитель и получатель не обязаны всегда доверять друг другу. Алгоритмы асимметричного шифрования помогают решить эти проблемы.

В алгоритмах асимметричного шифрования используется два ключа: открытый KU (Key pUblic) и закрытый KR (Key pRivate). Открытый ключ известен всем, закрытый держится в секрете. Алгоритмы и ключи должны удовлетворять следующим требованиям:

1. Легко вычислить пару (KU, KR).
2. Вычислительно легко зашифровать сообщение, используя открытый ключ: $C = E_{KU}(M)$.
3. Вычислительно легко расшифровать сообщение, используя закрытый ключ: $M = D_{KR}(C)$.
4. Вычислительно трудно, зная открытый ключ KU, найти соответствующий ему закрытый ключ KR.
5. Вычислительно трудно, зная зашифрованное сообщение C и открытый ключ KU, найти исходное сообщение M.
6. (*не для всех алгоритмов, для RSA) Открытый и закрытый ключи можно использовать в разном порядке: $E_{KU}(D_{KR}(M)) = M$.

Необходимо разобраться, что означают термины «вычислительно легко» и «вычислительно трудно». Они есть в теории алгоритмов, но там рассматривают, как ведет себя алгоритм в среднем случае, в худшем случае, нам же нужно, чтобы алгоритм был «вычислительно легким» или «вычислительно трудным» практически для всех входных значений, это более жесткое требование. Наконец, мы будем говорить, что задача является *вычислительно легкой*, если сложность ее решения пропорциональна полиному n^a, где n −длина входа. И задача является *вычислительно трудной*, если сложность ее решения пропорциональна 2^n, где n −длина входа.

Таким образом, криптография с открытым ключом основана на нахождении, так называемой, *односторонней функции*, которая в одну сторону $(y = f(x))$ вычисляется легко, а в обратную $(x = f^{-1}(y))$ – трудно, нет полиномиального решения обратной функции. В нашем случае задача немного отличается, у нас есть дополнительная зависимость от некого параметра k: $x = f_k^{-1}(y)$, когда k известно, вычисления обратной функции легкие, а если не известно, то задача вычислительно неразрешима.

Криптоанализ алгоритмов с обратным ключом

1. Лобовая атака.
   Единственная защита – использовать большую длину ключа. Необходим баланс между безопасностью и производительностью компьютера.

2. Атака на математическую функцию.
   Математики постоянно ищут новые атаки, например, сейчас кто только не решает задачу взлома алгоритма RSA. Будем говорить, что на сегодняшний день атака не найдена.

3. Вероятностное сообщение.
   Например, нам надо зашифровать ключ для DES длиной 56 бит: $C = E_{KU}(K)$. Нарушитель перебирает все возможные значения K, шифрует и сравнивает с зашифрованным сообщением.

Использование алгоритмов асимметричного шифрования

1. Шифрование и расшифрование.

   Отправитель передает получателю сообщение M. У получателя есть открытый KU_B и закрытый ключи KR_B, A знает открытый ключ B и то, что этот ключ принадлежит именно B. A шифрует сообщение, используя открытый ключ KU_B: $C = E_{KU_B}(M)$ и передает его B. Для того, чтобы расшифровать сообщение B использует свой закрытый ключ: $M = E_{KR_B}(C)$. Заметим, что KU_B и KR_B всегда являются парой ключей, они алгоритмически связаны, и на сегодняшний день не существует полиномиального алгоритма, которые позволяют, зная открытый ключ, вычислительным путем найти закрытый.

   Какие здесь есть сервисы: во-первых, это конфиденциальность, то есть никто кроме B не может получить исходное сообщение. Но у нас нет аутентификации отправителя, потому что мы предполагаем, что открытый ключ B знают все, таким образом, B не может определить, кто создал сообщение. Соответственно, нет целостности сообщения, так как его может изменить любой, кто знает открытый ключ B.

   Конфиденциальность, которую мы получаем здесь, принципиально отличается от той, которую мы получаем при использовании симметричного шифрования. В алгоритмах симметричного шифрования два участника знают общий секрет, если они доверяют друг другу в том, что ни одна сторона не передаст этот общий секрет кому-то еще, то неявно выполняется и аутентификация. В случае асимметричного шифрования аутентификации нет принципиально.

2. Создание и проверка цифровой подписи.

   Отправитель передает получателю сообщение M. У отправителя есть открытый KU_A и закрытый ключи KR_A, B знает открытый ключ A и уверен, что тот принадлежит именно A. Для создания подписи A использует свой закрытый ключ: $S = sign_{KR_A}(M)$ и передает B сообщение M и подпись S. Получатель B проверяет подпись, используя открытый ключ A: $M = ver_{KU_A}(S)$.

Из сервисов здесь имеются аутентификация отправителя, целостность сообщения, но принципиально нет конфиденциальности, и операцию расшифрования может произвести любой, кто знает открытый ключ A.

3. Обмен общим секретом (ключом).

   Встает вопрос, что делать, если нам нужна и конфиденциальность и аутентификация отправителя, и целостность. Нужно использовать двухстороннее асимметричное шифрование, чтобы и у отправителя и у получателя была пара из открытого и закрытого ключей. Участники знают открытые ключи друг друга. Если же им надо передать ключ, то можно вначале ключ подписать своим закрытым ключом, затем зашифровать результат открытым ключом B: $C = E_{KU_B}\left(sign_{KR_A}(K)\right)$. Можно в обратном порядке сначала зашифровать ключ, а затем подписать зашифрованный результат. С точки зрения криптостойкости разницы нет, однако с точки зрения удобства первый вариант разумнее, потому что можно хранить подпись к ключу и всегда проверять, что он не изменился. Далее этот ключ будет использоваться в алгоритме симметричного шифрования.

   Криптография с открытым ключом всегда связана с вычислительными трудностями, а у нас алгоритм вообще используется дважды. Зато в нашем случае отправитель и получатель могут не доверять друг другу.

В таблице 4.1 приведены примеры алгоритмов асимметричного шифрования и для чего они могут использоваться.

	Шифрование/расшифрование	Создание/проверка подписи	Обмен общим секретом
RSA	+	+	+
DSS/ГОСТ 3410	-	+	-
DH	-	-	+

Табл. 4.1. Алгоритмы асимметричного шифрования и их применение.

Алгоритм RSA

Алгоритм был разработан в 1977 году и назван по фамилиям его разработчиков Rivest, Shamir и Adleman. Пусть у нас есть сообщение M, и мы хотим зашифровать его следующим образом:

$$C = M^e (mod\ n),$$

а затем расшифровать:

$$M = C^d (mod\ n),$$

тогда открытым ключом будет $KU = \{e, n\}$, закрытым ключом – $KR = \{d, n\}$. Соответственно, нам надо найти взаимосвязь d, e и n, для того чтобы вышеуказанные выражения были верны для $\forall M < n$, и, следовательно, чтобы выполнялось соотношение:

$$M = M^{ed} (mod\ n).$$

В следующей лекции мы рассмотрим эту задачу подробнее, а сейчас я приведу уже результат, мы получим, что $n = p \cdot q$, где p и q – простые. Заметим, что n – это компонента открытого ключа, и ее знают все. (Чему равны компоненты e и d, разберем в следующей лекции.) Перед нами задача факторизации числа. Алгоритм начинается с выбора двух простых чисел, звучит легко, но это очень тяжело реализовать. Вообще говоря, теорема простого числа была доказана только в 2005 году, а алгоритм разработан в 1977 году. Мы выбираем большое число в нужном нам диапазоне (около 512 или 1024 цифр), и про него нужно сказать, является оно простым или нет. Если оно не является простым, то все те выкладки, которые мы будем рассматривать на следующей лекции, они окажутся неверны. Это не так просто определить, нужны специальные алгоритмы, теоремы. Существуют тесты, результатом которых является либо ответ «число не простое», либо «не известно». Вернемся к нашей задаче, если число не будет простым, то алгоритм окажется взламываем. С одной стороны, вы не найдете ни одного программного и аппаратного обеспечения, где не используется RSA, его рекомендует использовать NIST, но, с другой стороны, у него огромное количество недостатков, и криптографы недолюбливают этот алгоритм.

ЛЕКЦИЯ 5. КРИПТОГРАФИЯ С ОТКРЫТЫМ КЛЮЧОМ. АЛГОРИТМ ДИФФИ-ХЕЛЛМАНА

Математические утверждения, связанные с алгоритмом RSA

В предыдущей лекции мы начали рассматривать алгоритм RSA, поговорили о его свойствах, теперь займемся доказательством определенных утверждений. Напомню, что мы хотим найти такие e, d и n, чтобы $\forall M < n$ выполнялось соотношение

$$M = M^{ed}(mod\ n), \tag{5.1}$$

$KU = \{e, n\}$ назовем открытым ключом, $KR = \{d, n\}$ – закрытым ключом, зная d и n, невозможно найти e.

Утверждение 1.

Если a и n являются взаимно простыми, то есть НОД$(a, n) = 1$, то из того, что $a \cdot b = a \cdot c(mod\ n)$ следует, что $b = c(mod\ n)$.

Утверждение 2.

Множество всех чисел взаимно простых с p обозначим Z_p. Если p – простое, то $Z_p = \{1, \dots, p-1\}$.

Утверждение 3.

Для любого числа W в этом множестве Z_p существует единственное число W^{-1} такое, что $W^{-1}W = 1(mod\ p)$.

Утверждение 4.

Функция Эйлера $\varphi(p)$ – это количество чисел меньших p и взаимно простых с p. Если p – простое, то $\varphi(p) = p - 1$.

Утверждение 5.

Посчитаем, чему равна функция Эйлера, если взять два простых числа p и q и перемножить их $n = p \cdot q$. Числа, которые не являются взаимно простыми с n: $\{p, 2p, \dots, (q-1)p\}$, $\{q, 2q, \dots, (p-1)q\}$. Тогда

$$\varphi(p) = p \cdot q - \big((q-1) + (p-1) - 1\big) = (p-1)(q-1).$$

Теорема Ферма.

Если p – простое, то $A^{p-1} = 1(mod\ p)$. Рассмотрим числа $\{1, \dots, p-1\}$, умножим это множество на a по $mod\ p$: $\{1, \dots, p-1\} \cdot a(mod\ p)$, и получим $\{a\ mod\ p,\ 2a\ mod\ p, \dots,\ (p-1)a\ mod\ p\}$. Эти два множества совпадают с точностью до перестановки элементов. Перемножим элементы этих множеств между собой и получим равенство

$$(p-1)! = a^{p-1}(p-1)!\ mod\ p, \tag{5.2}$$

сократив его на $(p-1)!$, придем к требуемому соотношению

$$a^{p-1} = 1(mod\ p). \tag{5.3}$$

Теорема Эйлера.

Если A и n являются взаимно простыми ($НОД(A, n) = 1$), то $A^{\varphi(n)} = 1(mod\ n)$, где $\varphi(n)$ −функция Эйлера.

Опять же рассматриваем числа $\{a_1, \dots, a_{\varphi(n)}\}$ меньшие n и взаимно простые с n, умножаем это множество на А по $mod\ n$. Получим множества совпадающие с точностью до перестановки элементов, перемножим их, имеем

$$\prod_{i=1}^{\varphi(n)} a_i = A^{\varphi(n)} \prod_{i=1}^{\varphi(n)} a_i\ mod\ n, \tag{5.4}$$

сокращая произведение, придем к требуемому соотношению:

$$A^{\varphi(n)} = 1(mod\ n). \tag{5.5}$$

Наконец вспомним, что мы хотим найти такие e, d и n, чтобы $\forall M < n$ выполнялось соотношение $M = M^{ed}(mod\ n)$. Теперь мы знаем, что если M и n являются взаимно простыми: $НОД(M, n) = 1$, и n равно произведению двух простых чисел: $n = p \cdot q$, то $M^{(p-1)(q-1)} = 1(mod\ n)$.

Нам нужно, чтобы равенство выполнялось $\forall M < n$, а у нас пока есть только условие с взаимно простыми M и n. Пусть M и n не являются взаимно простыми: $НОД(M, n) \neq 1$, $n = p \cdot q$ и $M < n$. Тогда M кратно одному из простых чисел, например, $M = C \cdot p$. Следовательно, M является взаимно простым с q: $НОД(M, q) = 1$, и для него выполняется теорема Эйлера, то есть

$$M^{q-1} = 1(mod\ q). \tag{5.6}$$

Возведем выражение (5.6) в степень $p-1$, получим

$$M^{(q-1)(p-1)} = 1^{(p-1)}(mod\ q), \tag{5.7}$$

к сожалению, модуль пока не тот, который нужен. Перепишем (5.7) в виде

$$M^{\varphi(n)} = q \cdot l + 1, \tag{5.8}$$

это просто определение модуля числа, умножим (5.8) на $M = C \cdot p$, тогда

$$M^{\varphi(n)+1} = q \cdot l \cdot C \cdot p + M. \quad (5.9)$$

В итоге мы получим выражение

$$M^{\varphi(n)+1} = M(mod\ n), \quad (5.10)$$

а нам надо $M^{ed} = M(mod\ n)$. Отсюда следует, как мы должны выбирать e и d: они должны быть взаимно простыми с $\varphi(n)$, выберем e: $\text{НОД}\big(e, \varphi(n)\big) = 1$, а d – обратное к e по модулю $\varphi(n)$: $ed = 1\big(mod\ \varphi(n)\big)$.

Суть алгоритма RSA

Давайте подытожим алгоритм RSA, перечислим основные его составляющие:

1. $p,\ q$ – простые, выбираемые, закрытые
2. $n = p \cdot q$ – вычисляемо, открыто
3. $\varphi(n) = (p-1)(q-1)$ – вычисляемо, закрыто
4. e: $\text{НОД}\big(e, \varphi(n)\big) = 1$ – выбираемо, открыто
5. d: $ed = 1\big(mod\ \varphi(n)\big)$ – вычисляемо, закрыто

Тогда $KU = \{e, n\}$ – открытый ключ, $KR = \{d, n\}$ – закрытый ключ. Напомню, что мы шифруем сообщение путем преобразования $C = M^e(mod\ n)$, а расшифровываем – $M = C^d(mod\ n)$.

Вообще говоря, многие криптографы недолюбливают алгоритм RSA, считается, что задача факторизации числа для многих частных случаев разрешима, хуже того, могут быть плохие M и n, и число будет расшифровано. На самом деле, у алгоритма RSA много плохих корней, и на выбор p и q есть ограничения: они должны быть примерно одинаковой длины, $(p-1)$ и $(q-1)$ должны иметь очень маленький общий делитель и прочее. Тем не менее, алгоритм используется и рекомендуется к использованию NIST-ом. Этим алгоритмом нельзя шифровать, например, HTML-страницу, это вычислительно неразумно, поскольку у нас должно выполняться $M < n$. С помощью RSA мы можем зашифровать ключ, а большой трафик будем шифровать уже алгоритмами симметричного шифрования.

Пример.

Рассмотрим алгоритм RSA на простом примере.

1. Выберем два простых числа $p = 7$ и $q = 17$,

2. Вычислим их произведение $n = p \cdot q = 119$,
3. Вычислим функцию Эйлера $\varphi(n) = (p-1)(q-1) = 96$,
4. Вычислим e: $\text{НОД}(e, 96) = 1 \rightarrow\ e = 5$,
5. Вычислим d: $5d = 1(mod\ 96) \rightarrow\ d = 77$

Таким образом, имеем два ключа: $KU = \{5{,}119\}$ и $KR = \{77{,}119\}$. Предположим нам надо зашифровать сообщение $M = 19$, тогда зашифрованное сообщение будет

$C = 19^5(mod\ 119) = 66$. Получатель в свою очередь расшифрует сообщение и получит $M = 66^{77}(mod\ 199) = 19$.

Будем иметь ввиду, что приведенный пример абсолютно игрушечный, о таких числах как $p = 7$ и $q = 17$ речи не идет, обычно это числа, содержащие около 512 или 1024 цифры. Таким образом, алгоритм очень ресурсоемкий, но его плюс в том, что сколько ресурса тратится на шифрование, столько и на расшифрование.

Алгоритм Диффи-Хеллмана

Алгоритм получил название по фамилиям разработчиков, тоже основоположников криптографии криптографии с открытым ключом. Он основан на принципиально другой односторонней функции, это, так называемая, *задача дискретного логарифмирования*.

Пусть q – простое число, a называется *примитивным корнем* q, если множество $\{a\ mod\ q, a^2\ mod\ q, \dots, a^{q-1}\ mod\ q\}$ – это все числа от 1 до $q-1$ с возможными перестановками. Тогда задача, зная x, вычислить $y = a^x\ mod\ q$, вычислительно простая. Рассмотрим обратную задачу: решение для вещественных чисел – это обычный логарифм, а для целых чисел имеем задачу дискретного логарифмирования, для которой на сегодняшний день не известно полиномиальное решение. Для того, чтобы избежать атаки грубой силы, нужно выбирать большие значения q.

Перейдем к самому алгоритму Диффи-Хеллмана. У нас есть два участника I и J, они знают q и a. Участник I выбирает случайное число $x_i < q$ и вычисляет $y_i = a^{x_i}\ mod\ q$. То же самое делает участник J: выбирает случайное число $x_j < q$ и вычисляет $y_j = a^{x_j}\ mod\ q$. Поскольку x_i, x_j – случайные числа, у каждой стороны должен быть хороший генератор псевдослучайных чисел. Отметим также , что значения x_i, x_j являются закрытыми ключами, а y_i, y_j – открытыми, и, конечно, зная открытый ключ, вычислительно невозможно найти закрытый.

Итак, каждая сторона пересылает другой свои открытые значения y_i и y_j. Участник I получает значение y_j и возводит его в степень x_i:

$$K = {y_j}^{x_i} \; mod \; q = (a^{x_j})^{x_i} \; mod \; q = a^{x_j x_i} \; mod \; q, \tag{5.11}$$

аналогичные вычисления проводит участник J:

$$K = {y_i}^{x_j} \; mod \; q = (a^{x_i})^{x_j} \; mod \; q = a^{x_i x_j} \; mod \; q. \tag{5.12}$$

Таким образом, они получают одно и то же значение. Так как на сегодняшний день задача дискретного логарифмирования не решена, то зная открытые значения участников, невозможно вычислить K.

Тем не менее у алгоритма есть подводные камни, и нарушитель может кое-что сделать. Нарушитель может, определив свой собственный закрытый ключ $x_{\text{ОП}}$, вычислить свой открытый ключ $y_{\text{ОП}}$ по соответствующей формуле (a и q он знает), и в момент обмена открытыми значениями участников I и J он подменяет их открытые значения на свое, так, каждая сторона получает неправильное значение $y_{\text{ОП}}$ и вычисляет общий секрет с нарушителем. Так как алгоритм Диффи-Хеллмана используется для передачи секрета, который будет затем использоваться в алгоритмах симметричного шифрования, нарушитель остается на канале и расшифровывает весь трафик. Если участники I и J не аутентифицируют друг друга внешними алгоритмами по отношению к алгоритму Диффи-Хеллмана, то такая атака возможна. Но надо понимать, что обмен значениями между участниками надо еще и поймать, что технически может быть не просто.

Алгоритм Диффи-Хеллмана называют также *алгоритмом согласования ключа*, когда ни одна из сторон полностью секретный ключ не определяет. Второй способ переслать другой стороне общий секрет – это алгоритм RSA, так называемый, *алгоритм пересылки ключа*, в этом случае одна сторона полностью определяет секретное значение, шифрует его открытым ключом получателя, подписывает своим закрытым ключом и передает. Алгоритм согласования ключа считается более криптостойким и используется чаще, только необходимо дополнительно использовать аутентификацию.

Хэш-функции

Мы хотим для сообщения M произвольной длины создать некий «дайджест», его «отпечатки пальцев» – *хэш-код* $h = H(M)$, который будет иметь фиксированную длину. Хотелось бы, чтобы хэш-код был неким аутентификатором, чтобы он однозначно

определял сообщение. Но этого сделать, конечно, не получится, потому что мы отображаем большее множество на меньшее. Добьемся хотя бы того, чтобы было трудно найти разные сообщения с одним и тем же хэш-кодом.

Требования к хэш-функциям:

1. M – сообщение произвольной длины
2. h – фиксированной длины
3. $h = H(M)$ – вычислительно легко
4. Вычислительно невозможно найти M или его часть, зная h.
   Если данное требование выполнено, мы можем использовать хэш-функцию для обеспечения целостности. Пусть у нас есть отправитель A и получатель B, если они знают какой-то общий секрет S, то отправитель выполняет конкатенацию общего секрета и требуемого сообщения и пересылает получателю сообщение M и вычисленный таким образом хэш-код $h = H(S||M)$. Нарушитель видит M и h, но у него нет возможности, кроме как простым перебором, найти общий секрет. Тогда получатель берет M, выполняет конкатенацию общего секрета и полученного сообщения, вычисляет $h' = H(S||M)$, и сравнивает полученное и вычисленное значения хэш-кода: если $h' = h$ и хэш-функция криптографически сильная, можно считать, что сообщение не изменено.

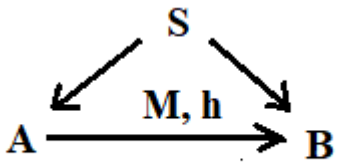

Рис. 5.1. Схема работы хэш-функции.

   Для обеспечения целостности не годится просто вычислить хэш-код как $h = H(M)$ и передавать получателю M и h, потому что нарушитель может заменить M на M', вычислить хэш-код $h' = H(M')$ и передать получателю. Поэтому нужен секрет, и необходимо как-то перемешать сообщение и секрет.

5. Для нашего сообщения M вычислительно трудно найти сообщение M': $H(M) = H(M')$.
6. Вычислительно трудно найти два сообщения M' и M'': $H(M') = H(M'')$.

Разберемся, в чем отличие 5-го и 6-го пунктов на примере. В пункте 5 у нас есть шарики с номерами: шар с номером x в руках и набор шаров в мешке с номерами y_1 ... y_k, номер может принимать значение от 1 до n. Сколько нужно перебрать шаров из мешка, чтобы найти с вероятностью $P(x = y_k) > \frac{1}{2}$ шар с тем же номером, что и у

нашего шара. Вероятность того, что мы вытащили два шарика с одинаковыми числами равна

$$P(x = y) = \frac{1}{n}, \tag{5.13}$$

а обратная вероятность

$$P(x \neq y) = 1 - \frac{1}{n}, \tag{5.14}$$

соответственно для y_k:

$$P(x = y_k) = 1 - \left(1 - \frac{1}{n}\right)^k > \frac{1}{2}. \tag{5.15}$$

Надо найти, при каком k выполняется (5.15). Используем приближение

$(1-a)^k \cong 1 - ka$, соответственно, $1 - \left(1 - \frac{1}{n}\right)^k \cong 1 - \left(1 - \frac{k}{n}\right) \cong \frac{1}{2}$, откуда $k \cong \frac{n}{2}$, то есть надо вытащить половину шариков.

Вернемся к нашим проблемам, пусть длина хэш-кода h будет равна n, соответственно, всего различных значений хэш-кода имеем 2^n, тогда с вероятностью больше $\frac{1}{2}$ мы через $\frac{2^n}{2}$ сообщений найдем сообщение с тем же хэш-кодом, что и у нашего сообщения. Отсюда возникает требование на длину хэш-кода – она должна быть большой, чтобы нарушителю было трудно подобрать другое сообщение с таким же хэш-кодом.

Рассмотрим задачу для 6-го пункта. Нам нужно найти в множестве из k элементов, каждый из которых принимает значение от 1 до n, два элемента с одинаковым значением, и чему должно быть равно k, чтобы $P(n,k) > \frac{1}{2}$. Мы можем выбрать k элементов $n(n-1)\dots(n-k+1)$ способами, чтобы не было повторений, всего у нас n^k способов, тогда вероятность того, что в наборе нет дублей равна $\frac{n(n-1)\dots(n-k+1)}{n^k}$, а искомая вероятность того, что дубли есть примет вид

$$P(n,k) = 1 - \frac{n(n-1)\dots(n-k+1)}{n^k} = 1 - \left(\frac{n-1}{n}\frac{n-2}{n}\dots\frac{n-k+1}{n}\right) =$$

$$= 1 - \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)\dots\left(1 - \frac{k-1}{n}\right) > \frac{1}{2}. \tag{5.16}$$

Воспользуемся неравенством $1 - x \leq e^{-x}$ в формуле (5.16) и получим

$$P(n,k) = 1 - e^{-\frac{1}{n}} \cdot e^{-\frac{2}{n}} \cdot e^{-\frac{k-1}{n}} \cong \frac{1}{2}.$$

Откуда $e^{\frac{k(k-1)}{n}} = 2$ и, следовательно, $k \cong \sqrt{n}$, то есть надо перебрать примерно $\sqrt{n}$ значений, чтобы найти два сообщения с одинаковым хэш-кодом. Это намного быстрее, чем в пункте 5.

ЛЕКЦИЯ 6. ПРИМЕРЫ КРИПТОГРАФИЧЕСКИХ ХЭШ-ФУНКЦИЙ

В прошлой лекции мы рассматривали хэш-функции, которые отображают сообщение произвольной длины в сообщение фиксированной длины. Мы определили 6 требований криптографически сильной хэш-функции. Напомню последние дыа требования:

5. Для нашего сообщения M должно быть вычислительно трудно найти другое сообщение M': $H(M) = H(M')$.

Мы подсчитали, что в среднем надо перебрать 2^{n-1} сообщение (n – длина хэш-кода), чтобы с вероятностью большей $\frac{1}{2}$ найти сообщение с таким же хэш-кодом.

6. Должно быть вычислительно трудно найти два разных сообщения M' и M'': $H(M') = H(M'')$.

Мы подсчитали, что в среднем надо перебрать $2^{\frac{n}{2}}$ сообщение (n – длина хэш-кода), чтобы с вероятностью большей $\frac{1}{2}$ найти два таких сообщения с одним и тем же хэш-кодом.

Поговорим о том, как использовать пятое требование. Мы будем подписывать не сообщение, а будем получать хэш-код от сообщения и подписывать его. Хэш-функции дают нам хэш-коды разумной длины: 128, 160, 256 бит, подписать их можно с помощью алгоритма RSA или DSS, это разумные по времени вычисления. Если нарушитель может составить другое сообщение с тем же хэш-кодом, то он подделал подпись, но сделать это не так-то просто.

Шестое требование более экзотическое, здесь нарушитель сам подбирает два разных сообщения с одинаковым хэш-кодом. Например, если есть некий сервис электронного нотариуса, то нарушитель может попробовать создать два сообщения с одним и тем же хэш-кодом, одно из них подписывает у нотариуса, второе посылает тому, кого хочет взломать, и посылает подпись, сделанную нотариусом.

Алгоритм MD5

MD5 – одна из первых хэш-функций, мы будем говорить о ней исключительно в историческом плане. На алгоритм уже более десяти лет назад найдена атака. Тем не

менее, MD5 практически везде реализована, мы не можем выбросить её. Длина хэш кода – 128 бит, это мало по современным меркам.

Разберемся, как устроен алгоритм. Сообщение разбивается на блоки по 512 бит, а последний блок добавляется таким образом, чтобы он тоже был кратным 512. Процедура добавления последнего блока стандартизована: в конец добавляется длина исходного сообщения, затем ставится единица, и остаток заполняется нулями.

Далее рассматриваются четыре 32-битных регистра, дающие в сумме 128 бит. Их начальные значения выбраны очень просто: $A = 0 \dots 7,\ B = 8 \dots F, C = F \dots 8, D = 7 \dots 0$. На вход функции MD5 подаются текущие значения регистров и очередной блок. Выходом являются новые значения регистра, которые затем подаются на вход той же самой функции и так далее. Значением хэш-кода является последнее значение этих четырех регистров.

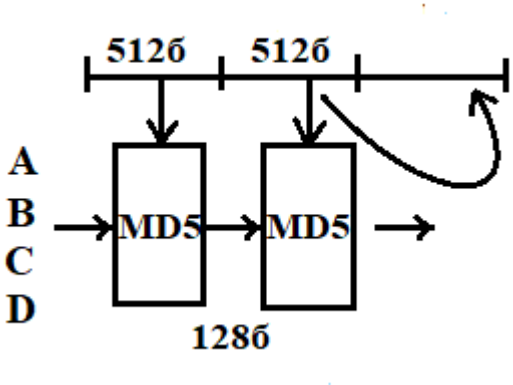

Рис. 6.1. Схема алгоритма MD5.

Разберемся, как устроена сама функция, преобразование MD5 (рис. 6.2). Используется 4 функции, на вход каждой подается выход предыдущей и очередной 512-битный блок сообщения. На выходе выполняется сложение по модулю 2^{32} значения регистра до преобразования и его выходного значения после всех преобразований, операция выполняется для всех четырех значений регистра. Каждая из 4-х функций на рис. 6.2. на самом деле является циклом длиной 16, в котором выполняются однотипные преобразования. Таким образом, каждый блок обрабатывается $4 \cdot 16 = 64$ раза.

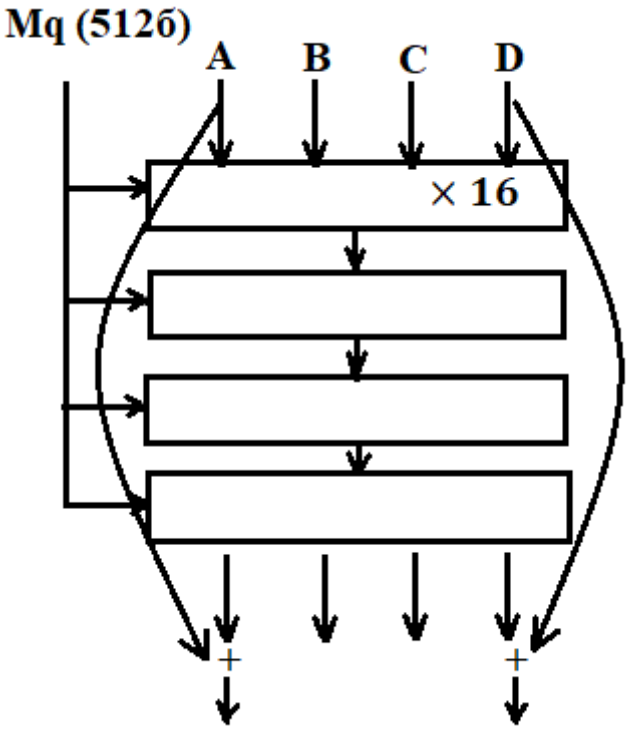

Рис. 6.2. Схема преобразования MD5.

Далее разберемся, что происходит в одной из этих функций:

$$A = B + CLS_S(A + f(B, C, D) + X[k] + T_i), \tag{6.1}$$

где CLS_S – циклический сдвиг влево на S бит, S зависит от номера цикла, $X[k]$ – k-е слово очередного блока, $T_i = 2^{32}\, abs(\sin(i))$ - некая константа, где i измеряется в радианах. Остальные преобразования получаются простым сдвигом:

$$B = A, C = B, D = C. \tag{6.2}$$

Функции f – это битовые функции над аргументами, их всего 4:

$$f_F = (B\&C)V(\bar{B}\&D), \tag{6.3}$$

$$f_G = (B\&D)V(C\&\bar{D}), \tag{6.4}$$

$$f_H = B \oplus C \oplus D, \tag{6.5}$$

$$f_I = C \oplus (B\&\bar{D}). \tag{6.6}$$

Наша заключается в том, чтобы максимально перемешать сообщение, без возможности восстановления.

Алгоритм SHA-1

SHA-1 (Secure Hash Algorithm version 1) широко используется, был принят в качестве стандарта NIST-ом в 1990 году. Длина хэш-кода 160 бит, то есть при прочих равных он более стойкий, в том числе к атакам типа «дня рождения»: для того чтобы найти два сообщения с одним и тем же хэш-кодом надо перебрать $2^{\frac{n}{2}}$ сообщений, где n – длина хэш-кода. День рождения можно рассматривать как своеобразный хэш-код, поскольку людей на Земле миллиарды, а дней в году всего 365. Вероятность того, что в какой-то группе найдется другой человек, у которого будет день рождения в один день с вами, довольно мала. Группа должна быть очень большой, чтобы в ней с вероятностью большей $\frac{1}{2}$ нашелся человек, у которого день рождения в тот же день. Но для того, чтобы в группе с вероятностью большей $\frac{1}{2}$ нашлось два человека, у которых день рождения в один день, группа должна быть достаточно маленькой. Это, так называемый, «парадокс дня рождения».

Изначально алгоритм похож на MD5, исходное сообщение разбивается на те же блоки по 512 бит, только теперь имеем уже 5 регистров вместо 4-х длиной 32 бита (SHA-1 и SHA-2 ориентированы на 32-битную аппаратуру). Начальные значения так же фиксированы, но немного другие. На вход функции подается 5 регистров и очередной блок, выходом являются новые значения регистров, которые подаются на вход точно такому же преобразованию вместе с новым блоком и так далее. Последнее значение регистров является хэш-кодом. Видим, что алгоритм SHA-1, действительно, очень похож на MD5. Одно из отличий заключается в том, что теперь у нас $4 \cdot 20 = 80$ циклов, а в MD5 их было $4 \cdot 16 = 64$.

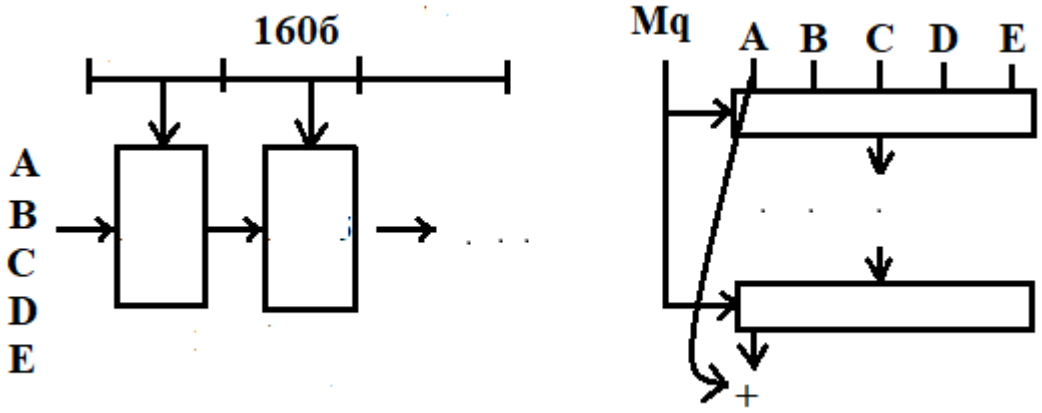

Рис. 6.3. Схема алгоритма SHA-1.

И сами преобразования теперь имеют несколько иной вид:

$$A = CLS_5(A) + f(B, C, D) + E + W_t + K_t), \quad (6.7)$$

где $W_t = W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3}$ при $t \geq 16$, константы $K_t = 2^{30}\sqrt{2},\ 2^{30}\sqrt{3},\ 2^{30}\sqrt{5},\ 2^{30}\sqrt{10}$, а остальные значения сдвигаются следующим образом:

$$B = A, C = CLS_{30}(B), D = C, E = D. \quad (6.8)$$

Функций f всего три, две из них (6.3) и (6.5) совпадают с алгоритмом MD5, к ним добавляется функция

$$f = (B\&C)V(B\&D)V(C\&D). \quad (6.9)$$

В начале 2000-х стало понятно, что длина хэш-кода 160 бит маловата. Начали находить атаки и на SHA-1, нужно было искать новые хэш-функции.

Алгоритм SHA-2

В 2001 году NIST принял новый стандарт SHA-2. (Хэш-функция хороша тем, что она быстрая, гораздо быстрее алгоритма симметричного шифрования, а криптография с открытым ключом самая медленная.) На самом деле SHA-2 содержит три функции: SHA-256, SHA-384 и SHA-512, где последние цифры обозначают длину хэш-кода (табл. 6.1). Длина блока составляет 512 бит, длина регистра (длина слова) – 32 бита. SHA-256 была разработана как самостоятельная хэш-функция, в которой перемешиваний и сдвигов было намного больше, поэтому она гораздо медленнее, чем SHA-1. SHA-512 не является новой функцией по сравнению с SHA-256, просто у нее увеличили в два раза длину блока и длину регистра при тех же преобразованиях. У SHA-384 такие же параметры, как и у SHA-512, при этом вычисляется SHA-512 и берутся младшие 384 бита. Отметим, что SHA-384 и SHA-512 уже ориентируются на 64-битную аппаратуру.

	Длина блока	Длина слова
SHA-256	512	32
SHA-384	1024	64
SHA-512	1024	64

Таблица 6.1. Хэш-функции SHA-2 и их свойства.

Алгоритм SHA-3

В 2012 году был разработан принципиально новый алгоритм хэширования, который уже имеет гораздо больше настроек и длину хэш-кода: 224, 256, 384 или 512. (Помним, что слишком большая длина хэш-кода не есть хорошо, так как нам нужно

этот хэш-код передавать, подписывать, это затратно.) SHA-3 был признан NIST-ом в качестве стандарта.

Алгоритм основан на «принципе губки». Исходное сообщение делится на блоки, используется начальный вектор, который состоит из двух частей r и c. Выполняется XOR r-части с первым блоком и результат подается в функцию f и так до тех пор, пока не закончится сообщение, это, так называемая, стадия «сжатия». Когда сообщение закончилось, начинается стадия «отжатия»: используется та же функция f, результат выдаем до получения хэш-кода нужной длины. Значения r и c настраиваются в зависимости от того быстрее или лучше нам необходимо хэшировать, также настраивается длина хэш-кода, и считается, что функцию f можно менять (авторы алгоритма предлагают несколько вариантов), но NIST принял одну функцию в качестве стандарта.

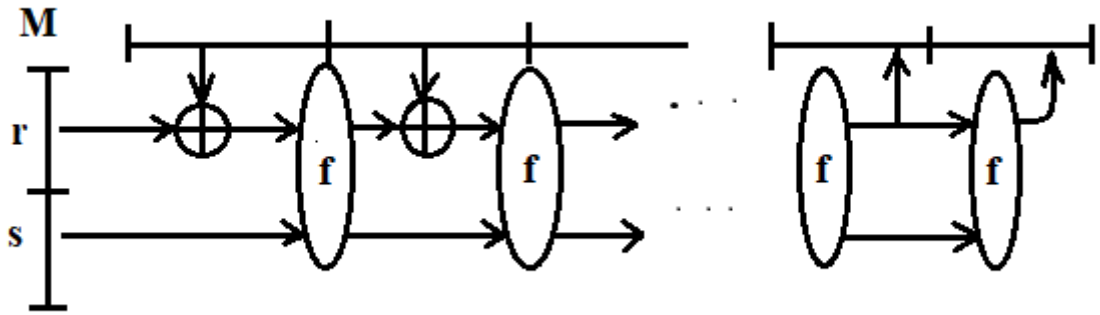

Рис. 6.4. Схема алгоритма SHA-3.

Алгоритм ГОСТ 3411

ГОСТ 3411 был принят в качестве стандарта в 1994 году, он не так популярен как алгоритмы симметричного шифрования. Алгоритм имеет довольно большой хэш-код длиной 256 бит, что вполне сравнимо с SHA-3. По сравнению с MD5 и SHA-1 алгоритм ГОСТ 3411 считается достаточно медленным. ГОСТ 3411 используется там, где обязуют использовать отечественный стандарт. Длина блока – 256 бит.

В ГОСТ 3411 сообщение разбивается на блоки длиной 256 бит, хэшируется с права налево. Есть стартовый вектор хэширования, если сравнивать с MD5 и SHA-1, у этих алгоритмов начальные значения регистров стандартизованы, в ГОСТ 3411 значение стартового вектора хэширования является параметром алгоритма. Далее, как обычно, вектор и текущий блок подаются на вход функции, а выход последнего преобразования является хэш-кодом.

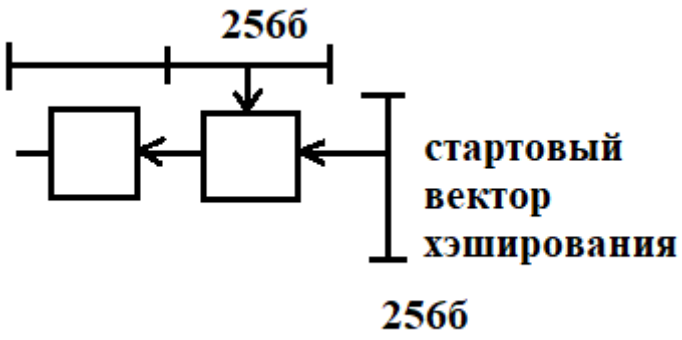

Рис. 6.5. Схема алгоритма ГОСТ 3411.

Рассмотрим, что происходит внутри функции:

1. Генерация ключей $K_1, \dots, K_4$.

   Здесь используются константы C_1, C_2, C_3, C_4 - это просто фиксированная последовательность нулей и единиц. Используется перестановка P: $y = \varphi(x)$, x и y – это порядковые номера байта в 256-битном блоке, поэтому $x, y \in 0, \dots, 31$, перестановка имеет вид:

   $$\varphi\big(i + 1 + 4(k - 1)\big) = 8i + k, \tag{6.10}$$

   где $i = 0 \div 3, k = 1 \div 8$. Фактически мы каждый байт переставляем в другое место по формуле (6.10). Также определен сдвиг A: $A(x) = x_1 \oplus x_2 || x_4 || x_3|| x_2$, где x_i – 64-битное значение. Далее определяются следующие значения: $U = h, V = M, W = U \oplus V,$ в итоге $K_1 = P(W)$ – наш первый ключ. Для следующих ключей K_i имеем $U = A(U) \oplus C_i, V = A\big(A(V)\big), W = U \oplus V$ и $K_i = P(W)$.

2. Шифрование с помощью алгоритма ГОСТ 28147.

   Производим шифрование h на ключах $h = h_4 || h_3 || h_2 || h_1$, где $||$ – это конкатенация. И шифруем $S_i = E_{K_i}(h_i)$, $S = S_1 || S_2 || S_3 || S_4$.

3. Перемешивание результата.

   Определим преобразование ψ: $\eta_{16} || \eta_{15} \dots || \eta_1 \to \eta_1 \oplus \eta_2 \oplus \dots \eta_{16} || \eta_{16} || \dots || \eta_2$ и преобразование $\varkappa(M, h) = \psi^{61}\left(\mathrm{h} \oplus \psi\big(M \oplus \psi^{12}(\mathrm{S})\big)\right)$.

Шифрование, кодирование и хэширование

Поговорим о терминах: шифрование, кодирование и хэширование, в чем между ними разница.

Шифрование – это преобразование блока данных с использованием ключа, то есть для того, чтобы зашифровать и расшифровать сообщение, нам необходим ключ.

Кодирование – это преобразование блока данных без использования ключа, таким образом преобразование из одной кодировки в другую может делать любой.

Хэширование – это преобразование, при котором принципиально невозможно восстановить исходное сообщение.

Обеспечение целостности сообщения

У нас есть отправитель и получатель, разберемся, что должны иметь, знать участники обмена, что должен сделать отправитель с сообщением, что должен проверить получатель, для того чтобы быть уверенным, что он получил именно то, что ему посылали. Контрольная сумма (разбиваем сообщение на блоки и выполняем продольный XOR) не защищает от преднамеренной подделки. Как нам защититься:

1. Цифровая подпись.
   У отправителя есть открытый и закрытый ключ, получатель должен знать открытый ключ отправителя. Если надо подписать сообщение длиной килобайт, мегабайт и больше, тогда отправитель хэширует сообщение, подписывает хэш-код своим закрытым ключом и отправляет получателю сообщение M и подпись для этого хэш-кода. В таком случае получатель может быть уверен в том, что он получил то, что посылали.

2. MAC (Message Authentication Code) – код аутентификации сообщения.
   Чаще всего используется код аутентификации сообщения, полученный с использованием хэш-функции HMAC. Просто так вычислить хэш-код и послать вместе с сообщением нельзя, целостности не будет. Соответственно, у отправителя и получателя должен быть общий ключ, тогда отправитель вычисляет хэш-код $h = HMAC(M, K)$ и отправляет его и M получателю. Здесь
   $$HMAC = H\left((K^{+} \oplus Opad)||H\big((K^{+} \oplus Ipad)||M\big)\right), \qquad (6.11)$$
   где константы $Opad =' 01011010'$, $Ipad =' 00110110'$. Это способ замешать ключ и сообщение.

ЛЕКЦИЯ 7. АЛГОРИТМЫ, ОСНОВАННЫЕ НА ЗАДАЧЕ ДИСКРЕТНОГО ЛОГАРИФМИРОВАНИЯ

Стандарт DSS

Нам осталось рассмотреть последний алгоритм асимметричного шифрования – DDS (Digital Signature Standard). Вообще говоря, DSS является стандартом, если мы имеем ввиду алгоритм, то нам следует называть его DSA (Digital Signature Algorithm). В нашей стране принят аналогичный стандарт – ГОСТ 3410.

На самом деле у нас уже есть один алгоритм, который умеет делать подпись – алгоритм RSA. Процедура выглядит следующим образом: у отправителя A есть открытый и закрытый ключи, соответственно, KU_A и KR_A, причем получатель B знает открытый ключ отправителя. Отправитель вычисляет хэш-код сообщения $h = H(M)$, подписывает его своим закрытым ключом $S = sign_{KR_A}(h)$ и отправляет сообщение M и подпись S получателю. Получатель, в свою очередь, вычисляет хэш-код от полученного сообщения $h' = H(M')$ и выполняет процедуру верификации с открытым ключом отправителя $S' = ver_{KU_A}(h')$, если полученный результат совпал с подписью отправителя $S' = S$, то считается, что сообщение было отправлено именно A и не было изменено.

Разберемся, как это реализовано в стандарте DSS. Начало похоже на алгоритм RSA: у отправителя A так же есть открытый и закрытый ключи: KU_A и KR_A, получатель B знает открытый ключ отправителя. Отправитель вычисляет хэш-код сообщения $h = H(M)$. (В отличие от алгоритма RSA, в DSS хэш-функция фиксирована – SHA-1). Далее отправитель подписывает хэш-код, используя свой закрытый ключ, глобальные параметры KR_G и случайное число k: $sign_{KR_A}(h, KR_G, k) = \binom{r}{s}$, в результате у него получаются два компонента подписи r и s. Отправитель передает получателю сообщение M и подпись в виде двух компонент. Получатель, в свою очередь, вычисляет хэш-код от полученного сообщения $h' = H(M')$ и выполняет процедуру верификации с открытым ключом отправителя $ver_{KU_A}(h', KR_G) = V$, у него получается компонента V, если $V = r$, то считается, что сообщение было отправлено именно A и не было изменено.

Таким образом, первое отличие алгоритмов RSA и DSS заключается в наличии у последнего случайного числа k. В стандарте DSS каждый раз, подписывая одно и то же сообщение одним и тем же ключом, мы получаем разные подписи. В связи с этим алгоритм RSA называют *детерминированным алгоритмом*, а DSS – *рандомизированным*.

Разберем более подробно стандарт DSS.

1. Определение глобального открытого ключа.

Начнем с глобального открытого ключа KR_G – его знают все: отправитель, нарушитель, получатель и мы с вами. Выбирается простое число p в диапазоне от 512 до 1024 бит, то есть длина p лежит в диапазоне: $2^{512} < p < 2^{1024}$. Выбирается простое число q, длина которого равна 160 бит: $2^{159} < q < 2^{160}$, причем q является делителем числа $p-1$, то есть $\frac{p-1}{q}$ – целое. Вычисляется число $g = h^{\frac{p-1}{q}} \, mod \, p$. Эти три числа $\{p, q, g\}$ и образуют глобальный открытый ключ KR_G .

2. Создание пары открытый, закрытый ключ.
   Далее надо определить закрытый и открытый ключи. В качестве закрытого ключа KR выбирается любое случайное число x меньшее q: $x < q$. Открытый ключ KU вычисляется как $y = g^x \, mod \, p$. Перед нами задача дискретного логарифмирования. Очередное отличие алгоритмов заключается в том, что они основаны на разных неразрешимых задачах: RSA на факторизации числа, DSS на дискретном логарифмированием. Считается, что задача дискретного логарифмирования более стойкая, а задача факторизации числа все-таки решается для многих частных случаев.

3. Создание подписи.
   Вычисляется хэш-код сообщения: $h = H(M)$. В отличие от алгоритма RSA, в DSS хэш-функция фиксирована – SHA-1. Выбирается случайное число $k < q$, вычисляются компоненты вектора:

$$r = (g^k mod \, p) mod \, q, \tag{7.1}$$

$$s = k^{-1}(H(M) + xr) mod \, q. \tag{7.2}$$

   Обратим внимание, что хэш-код сообщения входит в подпись линейно, а в алгоритме RSA мы возводили его в огромные степени. В DSS только внутри компоненты r есть операция возведения в степень, зависящая от случайного числа k. Тем не менее, подпись в стандарте DSS можно организовать довольно быстро. Минус заключается в том, что в результате можно ненароком получить DOS атаку.

4. Проверка подписи.
   Получатель проводит следующие вычисления:

$$w = s^{-1}(mod \, q), \tag{7.3}$$

   где s^{-1} – обратное число к s, то есть $ws = 1(mod \, q)$,

$$u_1 = H(M)w(mod \, q), \tag{7.4}$$

$$u_2 = rw(mod \, q), \tag{7.5}$$

$$V = (g^{u_1} y^{u_2} mod \, p) mod \, q. \tag{7.6}$$

Наша задача заключается в том, чтобы доказать равенство $r = V$. Мы видим, что этап проверки значительно дольше, чем этап создания подписи. Это одна из причин, по которым RSA все еще используется.

Тем не менее, государственные стандарты и стандарты, принятые NIST-ом основаны на задаче дискретного логарифмирования. Вообще говоря, все алгоритмы, использующие преобразования (7.1), (7.2), называются *семейством алгоритмов Эль-Гамаля*. (Алгоритмы отличаются тем, как случайное число k входит в формулы для компонент r и s.)

Вернемся к нашей задаче, нам необходимо доказать, что выполняется $r = V$.

Утверждение 1: $g^t mod\ p = g^{t\ mod\ q} mod\ p$
То есть степеней больших q не будет. Возведем g в степень nq и, учитывая, что $g = h^{\frac{p-1}{q}}\ mod\ p$, получим:

$$g^{nq} mod\ p = h^{\frac{n(p-1)q}{q}}\ mod\ p = 1^n (mod\ p) = 1,$$

здесь мы также учли теорему Эйлера:

$$h^{p-1} = 1 (mod\ p).$$

Тогда $g^t mod\ p = g^{nq\ +\ t\ mod\ q} mod\ p = g^{\ t\ mod\ q} mod\ p$, что и требовалось доказать.

Утверждение 2: $g^{\ a\ mod\ q + b\ mod\ q} mod\ p = g^{\ (a+b)\ mod\ q} mod\ p$
Это следует прямиком из первого утверждения.

Утверждение 3: $y^{(rw) mod\ q} mod\ p = g^{(xrw) mod\ q}\ mod\ p$
Это следует из определения y как $y = g^x\ mod\ p$.

Утверждение 4: $\big((H(M) + xr)w\big) mod\ q = k$
Напомню, что $s = k^{-1}(H(M) + xr) mod\ q$ и $w = s^{-1} (mod\ q)$, откуда получим: $(r \cdot s) mod\ q = kk^{-1}(H(M) + xr) mod\ q = (H(M) + xr) mod\ q$. В итоге имеем $\big((H(M) + xr)w\big) mod\ q = (k \cdot s \cdot w) mod\ q = k\ mod\ q = k$, здесь мы учли, что $(s \cdot w) mod\ q = 1$ и $k\ mod\ q = k$ при $k < q$.

Наконец получим итоговое равенство: $V = (g^{u_1} y^{u_2} mod\ p) mod\ q = \big(g^{H(M)w(mod\ q)} y^{rw(mod\ q)} mod\ p\big) mod\ q = \big(g^{(H(M)w + xrw) mod\ q}\ mod\ p\big) mod\ q = r.$

Поговорим больше о свойствах стандарта DSS. У нас есть 3 глобальных параметра: число g и 2 простых числа p и q, причем p значительно больше q, длина q равна длине хэш-кода, используемой функции, более того, число $\frac{p-1}{q}$ является целым.

Важно, что значения p, q и g всем известны. Далее создаем ключи: закрытый ключ x – случайное число меньшее q, открытый ключ – $y = g^x \ mod \ p$. Все знают y, но никто не может найти степень x. Далее мы подписываем сообщение, подпись рандомизирована, то есть в ней присутствует случайное число k. И сообщение линейно входит в подпись.

Алгоритм ГОСТ 3410

ГОСТ 3410 также принадлежит семейству Эль-Гамаля. Отличие этого алгоритма от DSS заключается в выборе q. Опять же p и q – простые числа, только теперь длина q равна 256 бит: $2^{255} < q < 2^{256}$, такой выбор объясняется тем, что у нас будет использоваться наша отечественная хэш-функция. Число $\frac{p-1}{q}$ так же должно быть целым. И число g определяется как $g = h^{\frac{p-1}{q}} \ mod \ p$. Задаем закрытый ключ как случайное число $x < q$, а открытый ключ как $y = g^x \ mod \ p$. Подпись рандомизировна: выбирается случайное $k < q$ и вычисляются

$$r = (g^k mod \ p) mod \ q, \tag{7.7}$$

$$s = (kH(M) + xr) mod \ q. \tag{7.8}$$

Как видим, запись компонента s отличается от той, которая была в алгоритме DSS. Соответственно, изменится процедура проверки подписи:

$$w = H(M)^{-1} (mod \ q), \tag{7.9}$$

$$u_1 = ws (mod \ q), \tag{7.10}$$

$$u_2 = (q - r)w \ mod \ q, \tag{7.11}$$

$$V = (g^{u_1} y^{u_2} mod \ p) mod \ q. \tag{7.12}$$

Если $V = r$, то подпись верна. Если при вычислении подписи какая-то компонента обратилась в ноль, берется другое значение k.

Итак, мы рассмотрели все основные алгоритмы, которые используются в криптографии с открытым ключом, три из них основаны на сложности решения задачи дискретного логарифмирования (Диффи-Хеллмана, DSS и ГОСТ 3410) и один основан на решения задачи факторизации числа (RSA). Во многих протоколах общий секрет вырабатывается с использованием алгоритма Диффи-Хеллмана, но чтобы избежать атаки (анонимный Диффи-Хеллмана) выполняется аутентификация с использованием подписи, созданной алгоритмом DSS. В этом случае может затрачиваться достаточно большое время на проверку подписи, по этому поводу NIST в рекомендациях пишет, что если такая ситуация вас не устраивает, то лучше используйте RSA.

Мы уже говорили о том, что алгоритм ассиметричного шифрования силен настолько, насколько неразрешима задача, лежащая в его основе: задача факторизации числа, задача дискретного логарифмирования. Эти задачи пытаются решать пытаются решать для многих частных случаев. Что делать, когда в один прекрасный день нас поставят перед фактом, что математики получили кучу премий, а мы с вами остались

без возможности организовать безопасную коммуникацию в интернете. Соответственно разрабатываются новые алгоритмы, ищутся новые односторонние функции. Наша задача найти такую функцию $y = f(x)$, которая в одну сторону решалась бы легко, а обратное преобразование $x = f^{-1}(y)$ было бы не известно. Один из вариантов решения проблемы – увеличение длины ключа, но таким образом мы создаем сложности и себе тоже.

Криптография на эллиптических кривых

Хорошо было бы найти новую одностороннюю функцию, которую никто не решил. Мы получим меньшую длину ключа при той же стойкости к атакам перебора, к известным частным случаям (плохие ключи) и, следовательно, большую производительность. Один из вариантов решения проблемы – использовать эллиптические кривые. Все вычисления будут вестись на эллиптической кривой, в общем случае ее уравнение принимает вид:

$$y^2 + axy + by = x^3 + cx^2 + dx + e. \quad (7.13)$$

В криптографии используется некий частный случай этой кривой:

$$y^2 = x^3 + ax + b\ (mod\ p). \quad (7.14)$$

У нас есть некоторые точки P и Q на нашей эллиптической кривой (рис. 7.1), наша задача – получить одностороннюю функцию, соответственно нам необходимо научиться складывать две точки $P + Q$, и удваивать точку $2 \cdot P$. После этого мы можем взять некоторую точку Q, умножить ее на k и получить новую $P = k \cdot Q$. Нам нужно, чтобы задача нахождения P, зная образующую точку Q, являлась простой, а задача нахождения коэффициента k, зная пару точек P и Q, не имела полиномиального решения.

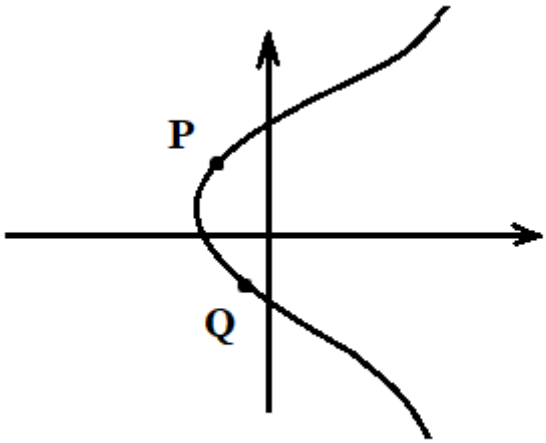

Рис. 7.1. Эллиптическая кривая.

Осталось только научиться складывать две точки. Пусть у нас на эллиптической кривой есть точка (x, y), тогда у нас есть и точка $(x, -y)$. Зададим бесконечно удаленную точку O, которая будет неким аналогом нуля, и предположим, что в ней сходятся все вертикальные прямые. Далее будем считать, что, если три точки, принадлежащие эллиптической кривой, лежат еще и на одной прямой, то их сумма равна этой бесконечно удаленной точке: $P + Q + S = O$. В этом случае сумма двух точек P и Q определяется через построение перпендикуляра: $P + Q = T$ (рис.7.2).

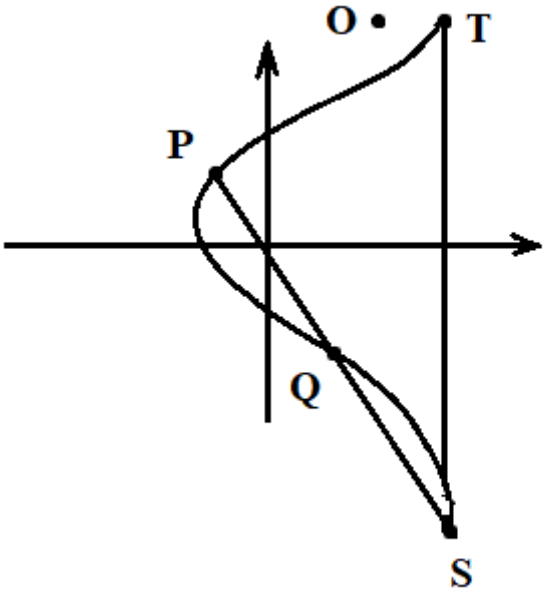

Рис. 7.2. Операция сложения двух точек эллиптической кривой.

Касательная в точке P на самом деле пересекает эллиптическую кривую в двух точках: $2 \cdot P + Q = O$. Соответственно, для того, чтобы удвоить точку нужно провести касательную до пересечения с третьей точкой эллиптической кривой и поднять или опустить перпендикуляр на ось x: $2 \cdot P = S$ (рис.7.3).

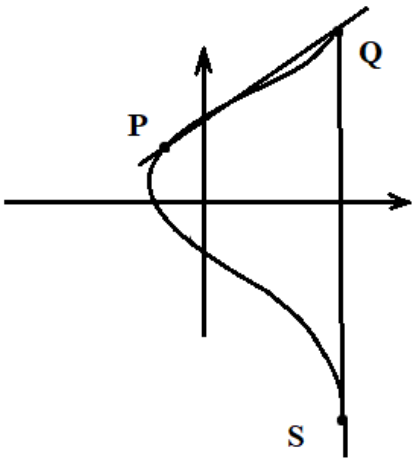

Рис. 7.3. Операция удвоения точки эллиптической кривой.

Таким образом, мы определили одностороннюю функцию. Мы можем взять некоторую образующую точку Q, умножить ее на k и получить новую точку P. Если же у нас есть две точки P и Q на эллиптической кривой, и мы хотим найти коэффициент k между ними, то эту задачу решить невозможно, по крайней мере пока она не решена, и мы можем использовать ее в своих целях. Отметим, что формула $P = k \cdot Q$ очень похожа на формулу дискретного логарифмирования, только там было возведение в степень, а у нас умножение, и числовую ось мы заменили на эллиптическую кривую. Плюс криптографии на эллиптических кривых в том, что мы имеем меньшую длину ключа, следовательно, большую производительность. Криптография на эллиптических кривых используется при создании цифровой подписи.

ЛЕКЦИЯ 8. ПРОТОКОЛЫ АУТЕНТИФИКАЦИИ И РАСПРЕДЕЛЕНИЯ КЛЮЧЕЙ

Криптография на эллиптических кривых

В предыдущей лекции мы начали рассматривать криптографию на эллиптических кривых – один из вариантов решения проблемы поиска односторонних функций. Эллиптическую кривую будем задавать в виде (7.14). Параметры a, b и p ($p -$ большое простое число) выбираются заранее, поэтому кривых может быть достаточно много. Рассмотрим поле над точками на эллиптической кривой. Мы уже научились складывать две точки и удваивать точку. Из чего мы получили следующую одностороннюю функцию: $P = k \cdot Q,$ где точка Q выбирается заранее и должна быть хорошей. Задача нахождения P, зная образующую точку Q, являлась простой, а задача нахождения коэффициента k, зная пару точек P и Q, не имеет полиномиального решения.

Можем использовать аналог алгоритма Диффи-Хеллмана. Напомним, в чем состоит алгоритм Диффи-Хеллмана. Он основан на дискретном логарифмировании на числовой прямой. Два участника выбирают случайное число x меньшее q, вычисляют $A^x mod\ q$ и посылают противоположной стороне. Далее участники возводят полученное значение в степень своего закрытого ключа и получают общий секрет, который кроме них никто не может вычислить (возможна атака man in the middle).

В нашем случае имеем эллиптическую кривую (7.14) с параметрами a, b и p, на этой кривой есть точка G. Каждая сторона выбирает случайное число, соответственно, n_I и n_J и вычисляет точку $P_I = n_I \cdot G$ и $P_J = n_J \cdot G$. Точки P_I, P_J являются открытыми, участники обмениваются ими, умножают на свой закрытый ключ и получают общую секретную точку: $K = n_I P_J = n_J P_I$.

Далее рассмотрим процедуру создания подписи на эллиптических кривых ECDSA (Elliptic Curve Digital Signature Algorithm), алгоритм принят в качестве стандарта NIST-ом. Вначале выбирается эллиптическая кривая $E_p(a, b)$, на ней должно быть много точек, количество точек делится на достаточно большое число n. На эллиптической кривой выбирается точка $P \in E_p(a, b)$. Далее создается пара открытого и закрытого ключей: выбирается случайное число $d < n$ и вычисляется точка $Q = d \cdot P$. Соответственно, точка d является закрытым ключом $KR = d$, а открытым ключом являются все параметры эллиптической кривой: $KU = \{Q, P, E_p(a, b), n\}$. Наконец, перейдем к созданию подписи: выбирается случайное число $k < n$, вычисляется точка $k \cdot P = (x, y)$, в качестве r берется координата x: $r = x(mod\ n)$. Если по какой-то причине эта точка нам не подходит, например, координата $x = 0$, то берется другое случайное число k. Вычисляется подпись

$$s = k^{-1}(H(M) + dr) mod\ n, \tag{8.1}$$

если $s = 0$, то опять же берется другое случайное число k. Процедура проверки подписи примет вид:

$$w = s^{-1}(mod\ n), \tag{8.2}$$

$$u_1 = H(M)w(mod\ n), \tag{8.3}$$

$$u_2 = rw\ mod\ n, \tag{8.4}$$

и вычисляется точка $u_1P + u_2Q = (x, y)$, если $x = r$, то подпись верна.

Криптография на эллиптических кривых позволяет шифровать открытым ключом получателя, но не большое сообщение, а, например, точку P_m. У нас есть эллиптическая кривая $E_p(a, b)$, на этой кривой есть некая образующая точка G, есть отправитель и получатель, у получатель имеет пару открытый и закрытый ключ: закрытый ключ – n_B и открытый – $P_B = n_B \cdot G$. Отправитель должен зашифровать точку P_m, зная открытый ключ получателя, он выбирает случайное число k, создает сообщение $C_m = \{P_m + k \cdot P_B,\ k \cdot G\}$ и отправляет его получателю с $k \cdot G$ в качестве подсказки. Получатель, в свою очередь, берет первую точку C_m, вычитает $n_B \cdot k \cdot G$ и получает исходную точку: $P_m + k \cdot P_B - n_B \cdot k \cdot G = P_m$.

Протоколы аутентификации и распределения ключей

Вернемся к симметричным алгоритмам шифрования, к обеспечению конфиденциальности. Мы говорили, что алгоритмы называются симметричными, потому что в них используется один и тот же ключ, который используют и отправитель и получатель. Теперь мы должны распределить этот ключ так, чтобы он был доставлен именно получателю, а не нарушителю, чтобы не было атак при выполнении этого обмена. Итак, у нас есть два участника A и B, пусть они используют ключ K_1. Через какое-то время они понимают, что зашифровали этим ключом достаточно большое количество информации или что используют его достаточно долгое время, поэтому им нужно сменить ключ. Нельзя старым ключом K_1 шифровать новый ключ K_2: $E_{K_1}[K_2]$, потому что в этом случае если у нас каким-то образом взломают ключ K_i, это будет означать, что взломаны все последующие и предыдущие ключи. В рассматриваемой нами схеме будут использоваться два типа ключей: *мастер-ключ* и *ключ сессии*. Трафик шифруется ключами сессии, а мастер-ключ используется исключительно для того, чтобы шифровать ключи сессии, то есть он должен быть более длинным, тщательнее храниться, должен использоваться в более сильном алгоритме шифрования. На самом деле может быть очень много участников, использовать один общий для всех ключ нельзя, для каждых двух участников должен быть свой ключ, соответственно, если у нас n участников, то нам понадобится $\frac{n(n-1)}{2} \sim n^2$ ключей. Мы до сих пор не обговорили, кто такие A и B, это могут быть отдельные компьютеры – хосты, но

гораздо чаще это приложения. В одном хосте может выполняться большое количество приложений, и в этом случае число n возрастает многократно. Соответственно, нам надо распределить порядка n^2 ключей, и мы помним, что их надо менять время от времени.

1. Ручное распределение. Администратор заходит на каждый хост или приложение и кладет в нужное место случайно сгенерированный ключ. Этот метод не масштабируемый, трудоемкий, поэтому полагаться на добросовестность администратора не всегда разумно.
2. Протокол.

У нас есть два участника, отправитель зашифровал ключ сессии мастер-ключом $E_{K_m}[K_s]$ и передал его другой стороне, нарушитель может попытаться заставить участников взаимодействовать с использованием старого ключа сессии. Мы предполагаем, что у нарушителя нет возможности взломать мастер-ключ, а только ключ сессии. Нарушитель может перехватить старую транзакцию, взломать ее, хотя бы простым перебором, и в следующий раз попытаться заставить участников использовать старый ключ сессии, это, так называемая, *replay-атака*.

Наша задача заключается в том, чтобы передать ключ тому, кому надо, иметь возможность его время от времени менять, и при этом защититься от replay-атаки. Для защиты от replay-атаки нам нужно каким-то образом пометить ключ, чтобы нарушитель мог отличить старый ключ от нового, например, можно просто пересчитывать ключи, но тогда получатся очень тяжелые протоколы.

1. Поэтому используется, так называемый, nonce – псевдослучайное число. B передает A nonce, A шифрует мастер-ключом ключ сессии и nonce: $E_{K_m}[K_s||nonce]$. B сравнивает полученный nonce с тем, который он отправлял, если эти два числа совпадут, то все хорошо.

2. Второй способ заключается в том, чтобы пометить ключ отметкой времени. A посылает B ключ сессии с отметкой времени: $E_{K_m}[K_s||T]$. Могут возникнуть следующие проблемы, во-первых, необходимо определить у получателя некое ΔT, в течение которого ключ будет считаться действительным. Существуют естественные сетевые задержки, поэтому если ΔT будет слишком маленьким, то мы будем отвергать действительные ключи, если ΔT будет слишком большим, то мы можем принять старый ключ. Вторая проблема состоит в том, что нарушитель может атаковать не протокол, а самих участников A и B, и попытаться переустановить у них неправильное время, тем самым заставив их

использовать старый ключ сессии. Далее при рассмотрении протоколов, которые используют отметку времени, мы будем неявно предполагать, что участники используют протокол синхронизации времени, обеспечивающий целостность.

Так как участников может быть много, нам нужна третья доверенная сторона, с которой участники имели бы общий ключ, ее называют KDC (Key Distribution Center). Задача всех этих протоколов заключается в том, чтобы распределить ключ к сессии между A и B, выполнить аутентификацию сторон и защититься от replay-атак (рис 8.1).

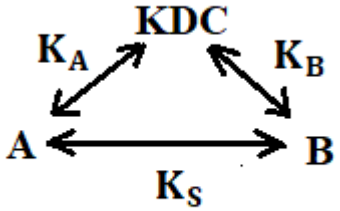

Рис. 8.1. Схема работы KDC, отправителя и получателя.

Ключи K_A и K_B определяются внешними по отношению ко всем этим протоколам способами, их количество пропорционально количеству участников n, таким образом мы ушли от n^2.

Мы предполагаем, что KDC абсолютно защищен всеми сетевыми, административными, организационными способами, потому что если у нас взломан KDC, то взломано все. Атаки могут производиться в сетевых взаимодействиях между участниками, участниками и KDC.

Протокол Нидхема-Шредера

Отправитель A посылает KDC свой идентификатор ID_A, идентификатор получателя ID_B и свой nonce N_A. В свою очередь, KDC возвращает A блок данных, зашифрованный ключом, который KDC разделяет с A: $E_{K_A}\left[K_s, ID_B, N_A, E_{K_B}[K_s, ID_A]\right]$, где $E_{K_B}[K_s, ID_A]$ – блок данных, зашифрованный ключом, который KDC разделяет с B. Далее A это расшифровывает с помощью ключа K_A и, таким образом, получает ключ сессии K_s, по ID_B проверяет, что блок создан для его взаимодействия с B, по N_A проверяет, что ключ не устарел, а блок данных $E_{K_B}[K_s, ID_A]$ участник A никак расшифровать не может – он посылает этот блок участнику B. Получатель B понимает,

что KDC создал этот блок в интересах B для A, аутентификация A выполнена. Дальше B должен выполнить свою аутентификацию: B посылает A свой nonce, зашифрованный ключом сессии: $E_{K_s}[N_B]$. И наконец участники должны убедиться, что они вычислили один и тот же секрет. A расшифровывает полученное сообщение, выполняет простейшее преобразование с N_B, зашифровывает результат и пересылает B: $E_{K_s}[f(N_B)]$. Перечислим коротко все этапы:

1. $A \to KDC: ID_A,\ ID_B, N_A$
2. $KDC \to A: E_{K_A}\left[K_s, ID_B, N_A, E_{K_B}[K_s, ID_A]\right]$
3. $A \to B: E_{K_B}[K_s, ID_A]$
4. $B \to A: E_{K_s}[N_B]$
5. $A \to B: E_{K_s}[f(N_B)]$

Как можно попытаться взломать этот протокол. Нарушитель может перехватить транзакцию номер 3, K_B он взломать не может, но при следующем взаимодействии он может попытаться заставить участников использовать старый ключ сессии, таким образом, подменить текущую транзакцию на старую: $E_{K_B}[K_s, ID_A] \to E_{K_B}[K_{s\,old}, ID_A]$. Это возможно в том случае, если A хранит старые ключи, поэтому во всех стандартах пишут сбросить старые ключи.

Протокол Деннинга

Протокол Деннинга уже использует отметку времени. Мы помним, что в этом случае у всех участников должны быть синхронизованы часы с использованием протокола, обеспечивающего целостность. Запишем коротко этапы данного протокола:

1. $A \to KDC: ID_A,\ ID_B$
2. $KDC \to A: E_{K_A}\left[K_s, ID_B, T, E_{K_B}[K_s, ID_A, T]\right]$
3. $A \to B: E_{K_B}[K_s, ID_A, T]$
4. $B \to A: E_{K_s}[N_B]$
5. $A \to B: E_{K_s}[f(N_B)]$

Протокол с использованием билета

Основная цель этого протокола заключается в том, чтобы, с одной стороны, использовать отметку времени, а, с другой стороны, избежать требования синхронизации часов участников. A будет проверять действительность ключа по

своему nonce N_A, а B – по отметке времени T_B, которую он ни с кем синхронизовать не должен.

Перечислим этапы протокола:

1. $A \to B: ID_A, N_A$
2. $B \to KDC: ID_B, N_B, E_{K_B}[ID_A,\ N_A, T_B]$
3. $KDC \to A: E_{K_A}[ID_A, ID_B, K_s, N_A], E_{K_B}[ID_A, ID_B, K_s, T_B], N_B$
4. $A \to B: E_{K_B}[ID_A, ID_B, K_s, T_B], E_{K_s}[N_B]$
5. $B \to A: E_{K_s}[N_B]$
6. $A \to B: E_{K_s}[f(N_B)]$

Блок данных $E_{K_B}[ID_A, ID_B, K_s, T_B]$ часто называют *билетом*.

Подумаем, как можно оптимизировать этот протокол, поскольку все перечисленные транзакции накладывают расходы. Если взаимодействие участников произойдет через достаточно короткий интервал времени, то есть они готовы не обращаться к KDC за новым ключом сессии и между собой использовать этот ключ сессии как новый мастер-ключ, то возможны следующие шаги:

$4'. A \to B: E_{K_B}[ID_A, ID_B, K_s, T_B], N_A$

$5'. B \to A: E_{K_s}[N_A]$

$6'. A \to B: E_{K_s}[f(N_A)]$

ЛЕКЦИЯ 9. ПРОТОКОЛ АУТЕНТИФИКАЦИИ KERBEROS

Протоколы аутентификации

В предыдущей лекции мы рассмотрели протоколы аутентификации с использованием симметричного шифрования. Теперь для тех же задач попробуем использовать криптографию с открытым ключом.

Аутентификационный сервер AS. Он будет выполнять только аутентификацию, не будет создавать общий секрет. Этапы:

1. $A \to AS: ID_A,\ ID_B$
2. $AS \to A: sign_{KR_{AS}}[ID_A, KU_A, T], sign_{KR_{AS}}[ID_B, KU_B, T]$
3. $A \to B: sign_{KR_{AS}}[ID_A, KU_A, T], sign_{KR_{AS}}[ID_B, KU_B, T], E_{KU_B}\left[sign_{KR_A}[K_s, T \text{ или } N_A]\right]$
4. $B \to A: E_{K_s}[N_B]$
5. $A \to B: E_{K_s}[f(N_B)]$

Участники должны знать открытый ключ AS, потому что иначе они не смогут проверить подписи, то есть что открытые ключи правильные.

KDC (Key Distribution Center). Этапы:

1. $A \to KDC: ID_A,\ ID_B$
2. $KDC \to A: sign_{KR_{KDS}}[ID_B, KU_B, T]$
3. $A \to B: E_{KU_B}[ID_A, N_A]$
4. $B \to KDC: ID_B, ID_A, E_{KU_{KDS}}[N_A]$
5. $KDC \to B: sign_{KR_{KDS}}[ID_A, KU_A, T], E_{KU_B}\left[sign_{KR_{KDS}}[K_S, N_A, ID_A, ID_B]\right]$
6. $B \to A: E_{KU_A}\left[sign_{KR_{KDS}}[\ \ ], N_B\ \right]$
7. $A \to B: E_{K_S}[N_B]$

Подпись в передаче ключа K_S (пункт 5) обеспечивает нам целостность. Алгоритм предусматривает существование ошибок проектирования со стороны участников, поэтому в нем присутствует некая избыточность, дублирование.

Протокол Kerberos

Протокол Kerberos наиболее известен, он был разработан в MIT для операционной системы UNIX, поддерживается и для W2K+. При разработке данного протокола стояла следующая задача: у нас есть большое количество рабочих станций,

за которыми сидят пользователи, и большое количество серверов, необходимо выполнить аутентификацию пользователя на сервер. Ее можно производить:

1. На рабочей станции.
   В этом случае все рабочие станции должны быть очень защищены, а это тяжело осуществить.
2. На сервере.
   С одной стороны сервер должен делать то, для чего предназначен, например, база данных – обрабатывать запросы, веб-сервер – создавать html-страницы, с другой стороны, сервер должен теперь еще и выполнять аутентификацию.

Оба эти варианта плохи так же тем, что в любой организации достаточно большой и долго функционирующей, как правило, всегда находится какой-нибудь старый забытый сервер. Проблема в том, что когда-то у пользователей были аккаунты на этом сервере, а мы любим создавать один и тот же аккаунт на все случаи жизни. С большой вероятностью аккаунты на старом сервере совпадут с актуальными.

Предлагается использовать KDC сервер, который состоит из аутентификационного сервера AS (Authentication Server) и сервера выдачи билетов TGS (Ticket Granting Server). Сервера AS и TGS нужно очень хорошо защищать. Отметим также, что хотелось бы, чтобы пользователь, который заходит на рабочую станцию не видел всех наших транзакций: он просто вводит логин и пароль и сидит работает. Перейдем к подробному рассмотрению работы протокола:

1. $C \to AS: ID_C, ID_{TGS}, TS_1$, где C – клиент, TS_i – отметка времени.
2. $AS \to C: E_{K_{C,AS}}\left[K_{C,TGS}, ID_{TGS}, TS_2, LT_2, Ticket_{TGS}\right]$, где блок $Ticket_{TGS} = E_{K_{AS,TGS}}\left[K_{C,TGS}, ID_C, AD_C, ID_{TGS}, TS_2, LT_2\right]$, LT_2 – время жизни ключа $K_{C,TGS}$ между клиентом C и TGS, AD_C – адрес клиента.

   Пункты 1 и 2 выполняются один раз на вход пользователя. Пользователь вводит пароль, клиентский модуль получает ключ $K_{C,AS}$ между клиентом и аутентификационным сервером, расшифровывает эту транзакцию, получает ключ $K_{C,TGS}$, информацию о времени действия этого ключа и билет на сервер TGS.

3. $C \to TGS: Ticket_{TGS}, Auth_C, ID_S$, где ID_S – индентификатор сервера, к которому клиент хочет получить доступ, $Auth_C = E_{K_{C,TGS}}[ID_C, TS_2]$ – аутентификатор клиента.
4. $TGS \to C: E_{K_{C,TGS}}\left[K_{C,S}, ID_S, ID_C, AD_C, Ticket_S\right]$, где $Ticket_S = E_{K_{S,TGS}}\left[K_{C,S}, ID_S, ID_C, AD_C, TS_3, LT_3\right]$ – билет на сервер.
   Пункты 4 и 5 выполняются один раз для каждого сервера.

5. C → S: $Ticket_S, Auth_C$, где $Auth_C = E_{K_{C,S}}[ID_C, TS_4]$.

   К этому моменту клиент аутентифицировал себя на сервере, теперь сервер должен аутентифицировать себя клиенту.
6. $S \rightarrow C$: $E_{K_{C,S}}[TS_4 + 1]$

   С одной стороны, мы хотим, чтобы общий секрет был только у двух участников, с другой стороны, мы не хотим полагаться на добросовестность пользователя и надеяться, что он будет создавать на каждый сервер свой пароль, и не хотим постоянно на каждой транзакции запрашивать у пользователя новый пароль. Следующая наша задача: не допустить, чтобы пароли гуляли в открытом виде по сети, для этого мы пароль используем в качестве ключа шифрования. В алгоритме реализована защита от replay-атак.

Рассмотрим эталонную модель работы пользователя, какие транзакции при этом возникают. Пользователь включает свой компьютер, в течение своего рабочего дня он должен сначала посмотреть почту, зайти на корпоративный веб-сайт, в БД, снова посмотреть почту и зайти на ftp-сервер. Вначале выполняются транзакции 1 и 2, то есть выполняется аутентификация пользователя, и он получает билет на TGS. Далее идут транзакции 3, 4: пользователь получает билет на почтовый сервер, и транзакции 5, 6: он получает доступ к своей почте. Следующим шагом пользователю нужно зайти на корпоративный веб-сервер, значит выполняются транзакции 3, 4 – он обращается к TGS за билетом на веб-сервер, и 5, 6 – он получает доступ к веб-серверу. Доступ к БД осуществляется за счет транзакций 3, 4, 5, 6. Для повторного доступа к почте пользователь использует полученный ранее билет, поэтому здесь остаются только транзакции 5 и 6. И наконец для доступа к ftp-серверу совершаются транзакции 3, 4, 5, 6. Заметим, что при повторном подключении билет используется один и тот же, а аутентификатор – каждый раз разный, тем не менее у билета есть время жизни, аутентификатор же является одноразовой конструкцией, это и есть защита от replay-атаки.

У нас есть аутентификационный сервер AS и некоторое количество клиентов n, для каждого нужно завести на сервере логин и пароль, в таком случае усилие администратора пропорционально n. Также у нас есть TGS, AS и TGS должны иметь общий секрет вне рамок нашего протокола. Если мы имеем дело с большой корпорацией, то возможно существование нескольких TGS, каждый из них должен иметь общий секрет с AS. Далее идут сервера (m штук), они должны, в свою очередь, иметь общий секрет с TGS. В итоге получаем $m + n$ действий, которые администратору нужно сделать «вручную», вне протокола распределить $m + n$ общих секретов.

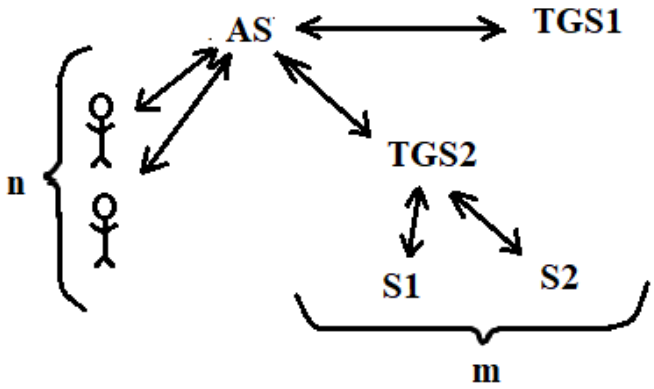

Рис. 9.1. Схема протокола Kerberos.

Ключ $K_{C,S}$ в дальнейшем можно рассматривать как мастер-ключ и использовать его для шифрования транзакции.

У нас есть AS, TGS, некие совокупности рабочих станций WS и серверов S – все это называется *областью* (realm). Таких областей может быть много, могут существовать так называемые, удаленные области (remote realm). Пользователю может понадобиться доступ на удаленный сервер, который не против предоставить ему доступ, если пользователь будет аутентифицирован в своей области. Для этого нужна аутентификация двух удаленных TGS. Протокол увеличится на две транзакции, после транзакций 1, 2 проводится аутентификация на удаленный TGS, и наш TGS получает билет на удаленный. Возникает, так называемое, *доверие* между областями, мы можем строить различные графы, деревья по взаимодействию между областями с использованием протокола Kerberos.

Протокол Kerberos был разработан в конце прошлого века, существует в двух версиях Kerberos v.4 и Kerberos v.5. Мы рассматривали версию v.4, поговорим о ее недостатках. Изначально протокол Kerberos разрабатывался, когда единственным алгоритмом симметричного шифрования был DES, из пароля пользователя создавался 56-битный ключ шифрования. Никаких попыток договориться, что использовать для шифрования, не предпринималось. Мы увидим, что в следующих протоколах в первых транзакциях участники будут договариваться, какой алгоритм использовать для шифрования. Вообще говоря, аутентификация с использованием пароля считается слабой в отличие от аутентификации с помощью цифровой подписи, потому что существуют, так называемые, *словарные атаки* или *атаки по словарю*. Есть две политики создания паролей: первая заключается в том чтобы позволить пользователю использовать в качестве пароля все, что угодно. При словарной атаке берется множество всех возможных паролей и перебирается, это гораздо быстрее, чем даже взломать алгоритм RSA. Второй вариант: администратор дает пользователю случайный

набор символов в качестве пароля, конечно, такой пароль будет не запоминаемым, поэтому он обычно записывается в блокнот и кладется около компьютера – это тоже не пароль. Другая проблема версии v.4 – это время жизни ключа, поскольку иногда бывает нужна аутентификация для долгих периодических работ, когда пароль ввести некому, например, работа оставлена на ночь.

В версии v.5 были добавлены опции, которые клиент запрашивает у аутентификационного сервера. С помощью этих опций можно выбрать какой будет использоваться алгоритм шифрования, например, с сегодняшнего дня мы переходим на такой-то алгоритм. Была введена предаутентификация с использованием асимметричного шифрования, в частности цифровой подписи. Это стало особенно популярно, когда появились смарт-карты (внешне похоже на флэшку), на которые можно положить открытый, закрытый ключ, на них осуществлять создание подписи, это безопаснее, чем на рабочей станции. Появились билеты, которые могут обновляться и использоваться для долговременных работ. Таким образом, версия v.5 позволяет более гибкую аутентификацию. Осталась проблема: первым выполняет аутентификацию клиент, а это считается не самым правильным решением – отправлять свои данные неизвестно кому, во всех протоколах, которые мы будем рассматривать далее в курсе, первым будет идентифицировать себя сервер.

ЛЕКЦИЯ 10. ИНФРАСТРУКТУРА ОТКРЫТОГО КЛЮЧА

Алгоритмы с открытым ключом

Вернемся к рассмотрению криптографии с открытым ключом, а после и инфраструктуры открытого ключа или PKI (Public Key Infrastructure). В этих алгоритмах есть два ключа: открытый и закрытый. Основное свойство алгоритмов с открытым ключом заключается в том, что зная открытый ключ, вычислительно не возможно найти соответствующий закрытый ключ. В результате мы можем создавать цифровую подпись. Еще одна возможность, зная открытый ключ получателя, мы можем зашифровать сообщение и быть уверенными, что расшифровать его сможет только тот, кто знает закрытый ключ, то есть получатель. Зашифровать сообщение может кто угодно, поскольку для этого используется открытый ключ. Поэтому алгоритмы с открытым ключом и подвержены атаке man in the middle. Нарушитель не может узнать чужой закрытый ключ, но может создать свою пару открытого и закрытого ключей и подменить открытый ключ законного участника на свой. Происходит нарушение целостности, нам нужно аутентифицировать открытый ключ. Мы уже рассматривали два способа обеспечения целостности: с помощью общего секрета у участников и цифровая подпись. Первый вариант нам не подходит, поскольку придется обеспечивать общий секрет каждого с каждым, таким образом, мы заменяем одну проблему на еще более сложную. Мы будем использовать цифровую подпись, нам необходимо подписать идентификатор A и его открытый ключ KU_A неким доверенным органом – сертификационным центром CA (Certificate Authority). Третья доверенная сторона должна подписать своим закрытым ключом идентификатор, открытый ключ и какую-то дополнительную информацию (все вместе называется сертификатом открытого ключа) того, кого она удостоверяет.

Сертификат X.509

На самом деле форматов сертификатов очень много, но наиболее распространенный – это сертификат X.509 v.3. Поговорим о том, откуда пошли такие названия. Существует организация ISO, которая выпускает стандарты во многих направлениях, самая известная разработанная ей группа стандартов – это OSI, также был разработан набор стандартов на взаимодействие распределенных серверов – X.500. Стандарт X.509 изначально был частью X.500, он отвечает за аутентификацию взаимодействующих сторон. Версия X.509 v.3 была принята в качестве стандарта IETF (выпускает стандарты rfc – request for comment).

Перейдем к рассмотрению сертификата X.509 v.3, мы имеем следующие поля:

- версия

- серийный номер
- имя сертификационного центра
- имя субъекта, для которого выпущен сертификат
- период действительности сертификата
- открытый ключ субъекта и его возможные параметры

Все это подписывается закрытым ключом сертификационного центра. Отметим, что весь этот блок нельзя изменить, это может сделать только сертификационный центр, и поля всем известны, то есть туда нельзя положить ключ. Итак, поля, которые мы перечислили, были в первой версии сертификата, почти сразу после ее выпуска, стало понятно, что список хочется дополнить. Соответственно, во второй версии появились дополнительные поля:

- идентификатор сертификационного центра – ID CA
- идентификатор субъекта

Обратим внимание, что у нас очень много различных типов субъектов, которым требуется создавать подписи: человек, маршрутизатор и т.п. В третьей версии уже было введено понятие:

- расширения

Расширения позволяют записать какие-то дополнительные характеристики субъекта, окружения и так далее. Расширений много, они могут быть самые разные, их можно как включать в сертификат, так и не включать – это необязательное поле. Оно имеет следующий формат:

$$\langle \text{имя} \rangle = \langle \text{значение} \rangle \langle \text{признак критичности} \rangle,$$

где признак критичности принимает значения либо true, либо false. Если расширение критично (true), то это означает, что проверяющая сторона должна понимать семантику данного расширения. Если же проверяющая сторона не понимает семантики конкретного расширения, которое положено в сертификат, то есть там где-то стоит неверное значение для этого расширения, то она должна считать данный сертификат недействительным. В случае когда признак критичности равен false, и проверяющая сторона не понимает семантики конкретного расширения, то она может на свое усмотрение считать или не считать расширение действительным.

Цель инфраструктуры открытого ключа – предоставить действительный доверенный открытый ключ участника. Сертификационный центр должен отслеживать сертификат в течении всего времени его жизни. Когда у нас есть такой сертификат, то:

1. Любой участник, у которого есть открытый ключ сертификационного центра KU_{CA} может получить открытый ключ субъекта $KU_{\text{суб}}$, для которого создан данный сертификат.

2. Никто кроме сертификационного центра не может что-либо изменить в сертификате и, соответственно, создать другой сертификат с другими свойствами.

Сертификат, созданный сертификационным центром для участника A, будем обозначать $CA \ll A \gg$. У нас не может быть одного сертификационного центра на весь интернет, потому что, во-первых, это большая нагрузка, а, во-вторых, мы не обязаны доверять сертификационным центрам, которые расположены неизвестно где. Таким образом, сертификационных центров может быть много. Если участник A пришлет свой сертификат $CA_1 \ll A \gg$, выпущенный CA_1, участнику X, который обслуживается другим центром CA_2, то он окажется для него бесполезен, потому что он не знает открытого ключа CA_1 (рис. 10.1).

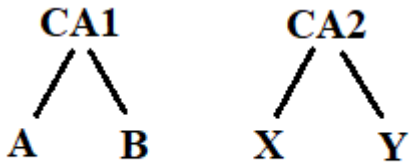

Рис. 10.1. Схема передачи сертификата между участниками разных сертификационных центров.

В этом случае есть следующие варианты действий:

1. Создать головной сертификат CA_0, который подпишет открытые ключи сертификационных центров (рис. 10.2). Получаем, так называемую, *цепочку сертификатов*, то есть субъект предыдущего сертификата является подписывающей стороной в следующем сертификате:
   $CA_0 \ll CA_1 \gg CA_1 \ll A \gg$. Если у участника есть открытый ключ головного центра, то он проверяет сначала сертификат $CA_0 \ll CA_1 \gg$, получает из него открытый ключ CA_1, затем проверяет сертификат $CA_1 \ll A \gg$ и из него получает открытый ключ конечного участника A.

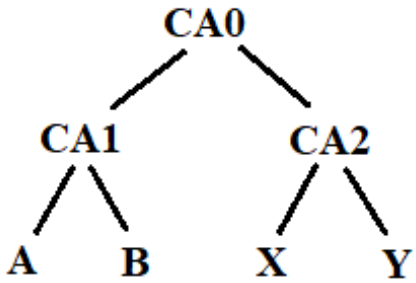

Рис. 10.2. Цепочка сертификатов.

2. Второй способ называют *кросс-сертификацией*, сертификационные центры подписывают открытые ключи друг друга (рис. 10.3). Это тоже цепочка сертификатов: субъект предыдущего сертификата является подписывающей стороной в следующем сертификате: $CA_2 \ll CA_1 \gg CA_1 \ll A \gg$. Если у участника X есть открытый ключ CA_2, то он сначала проверяет сертификат $CA_2 \ll CA_1 \gg$, получает из него открытый ключ CA_1, затем проверяет сертификат $CA_1 \ll A \gg$ и из него получает открытый ключ конечного участника A. Отметим, что цепочка может быть любой длины.

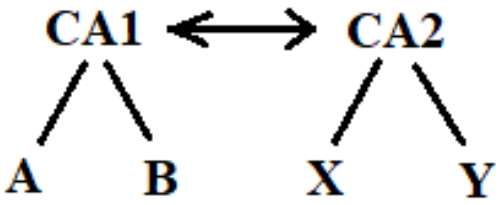

Рис. 10.3. Кросс-сертификация.

Оба способа могут применяться совместно, комбинироваться. Итак, для проверки такой цепочки проверяющая сторона должна знать открытый ключ головного центра, но почему мы должны ему доверять. Головной сертификат активно используется в браузерах, при установке браузера в нем уже зашиты корневые сертификаты, таким образом, мы доверяем цепочке сертификатов настолько, насколько мы доверяем той или иной реализации браузера.

Сертификат CRL

В сертификате у нас стояло две даты, определяющие время, в течении которого сертификат считается действительным. На самом деле в этот период времени может что-то произойти, например, потерян закрытый ключ, соответствующий открытому ключу, для которого выпущен сертификат. В подобных ситуациях нужно срочно что-то предпринимать, а именно отменить сертификат. Это также может потребоваться в случае ухода офиса в отпуск: мы хотим подстраховаться, гарантировано обезопаситься от взлома. Для этого существует сертификат CRL (Certificate Revocation List – список отмененных сертификатов), каждый сертификационный центр через определенные промежутки времени ΔT обязан выпускать список отмененных сертификатов. Формат списка имеет примерно вид: заголовок, содержащий дату и причину отмены, серийные номера отмененных сертификатов (SN_1, SN_2 и т.д.) и подпись выпускающегося CRL.

Период ΔT прописан в регламенте конкретного сертификационного центра, он зависит от важности выпускаемых сертификатов. Чем больше ΔT, тем хуже проверяющая сторона следит за отменой сертификатов, чем ΔT меньше, тем больше нагрузка и на проверяющую сторону, и на сертификационный центр, и на трафик.

Архитектура PKI

Основным участником является, так называемый, субъект или конечный участник (EE – End Entity), для которого выпускается сертификат, это может быть человек, сервер, маршрутизатор. Другой участник процесса – сертификационный центр CA, который выпускает сертификат и отслеживает его в течении всего времени жизни сертификата. Сертификационный центр должен быть абсолютно защищен, если его взломают, то придется отменять и перевыпускать все выпущенные им сертификаты. Поэтому если у нас большой сертификационный центр, который выпускает сертификаты для большого количества участников, то может существовать регистрационный центр RA (Registration Authority), который выполняет функцию взаимодействия с субъектами, а сертификаты выпускает сертификационный центр. Регистрационный центр ставится максимально удобно для субъектов, а сертификационный центр ставится максимально защищенным. Далее у нас есть выпускающий CRL, возможно также взаимодействие с другими удаленными сертификационными центрами для выполнения кросс-сертификации. Наконец должно быть хранилище сертификатов и CRL, к которому получают доступ субъекты, и сертификационный центр кладет туда и получает оттуда сертификаты. И есть проверяющая сторона RP (Relying Party), то есть та сторона, которая полагается на инфраструктуру открытого ключа и получает оттуда открытые ключи субъектов. Примером проверяющей стороны является браузер.

Поговорим о том, какие есть требования к безопасности каждого из участников. Самой защищенной структурой должен быть сертификационный центр, он, например, должен иметь усиленный вариант операционной системы, различные межсетевые экраны и прочее. А вот к хранилищу сертификатов и CRL требования по безопасности в принципе отсутствуют, потому что и сертификат, и CRL являются самозащищенными структурами данных: во-первых, там нет никаких секретов (ключей), а, во-вторых, это структуры данных, которые подписаны, то есть в них нельзя ничего изменить без обнаружения. В результате хранилище сертификатов и CRL должно быть максимально доступно.

Способы хранения информации

Наше хранилище сертификатов и CRL, с одной стороны, должно быть максимально распределенным, а с другой – максимально стандартизованным. Перечислим, какие существуют способы хранения информации:

1. Файловая система
   Мы можем положить сертификат как структуру данных в файл, к тому же файловая система может быть распределенной, но на способы именования файловой системы не существует стандартов, то есть мы можем положить данные, но никто не узнает, куда мы это все положили, поэтому этот вариант нам не очень подходит.

2. Реляционные базы данных
   Мы можем положить сертификат как базу данных, но минус такого способа хранения информации заключается в том, что опять же у нас нет стандартной схемы базы данных, то есть как мы назовем те или иные таблицы, столбцы, какие мы установим между ними соотношения – это дело разработчика конкретной базы данных.

3. LDAP/AD
   Возвращаемся к стандартам X.500, разработанным ISO, одним из стандартов там был протокол доступа к директории DAP. Однако он получился тяжелым, трудно реализуемым, и о нем забыли, и уже позже его модифицировали в облегченный протокол LDAP (Lightweight Directory Access Protocol) и, конечно, тут же начали утяжелять, так появился протокол AD (Active Directory). Это древовидная структура данных DIT (Directory Information Tree), где каждый узел, с одной стороны, представляет собой совокупность записей, а с другой стороны, содержит поддерево. Отметим, что структура записей стандартизована, существует описание структуры DIT, которое называется Core, там сказано в каком узле какие записи обязаны присутствовать, могут присутствовать, какой диапазон значений может быть у каждой записи.

Мы хотим создать инфраструктуру открытого ключа которая охватывала бы фактически весь интернет, нам нужно, чтобы проверяющая сторона и субъект общались на одном языке, общими терминами. В качестве имени сертификационного центра в сертификат записывается DN (Distinguish Name) CA, имени субъекта – DN EE.

Проверка действительности сертификата

Проверяющая сторона получает некий сертификат и должна определить действительный он или нет, для этого она выполняет следующий порядок действий:

1. Проверка цифровой подписи (в смысле математического алгоритма)
2. Проверка значений всех полей, указанных в сертификате
3. Проверка CRL, необходимо запросить список отмененных сертификатов и проверить, не отменен ли данный сертификат

ЛЕКЦИЯ 11. КРИПТОГРАФИЧЕСКИЕ ПРОТОКОЛЫ

На прошлой лекции мы рассматривали архитектуру открытого ключа. Мы поняли какие уязвимости существуют у открытого ключа, с одной стороны, математики гарантируют, что никто не сможет найти закрытый ключ, однако нарушитель может осуществить подмену открытого ключа (атака man in the middle). Поэтому введено понятие сертификационного центра, который обеспечивает целостность связки идентификатора участника и его открытого ключа. Нам интересен только один способ обеспечения целостности – цифровая подпись. Соответственно, у нас много форматов сертификатов, но самым распространенным является X.509 v.3. Мы рассмотрели его формат, понятие цепочки сертификатов, мы поняли, что нам необходим еще один механизм, который позволит нам отменить сертификат раньше срока, так называемый, CRL. Также мы рассмотрели протокол LDAP как предпочтительное хранилище сертификатов и CRL, поговорили о том почему оно является предпочтительным. Когда мы говорили о формате сертификатов X.509, мы перечисляли поля:

- version
- SN – серийный номер
- CA – имя сертификационного центра
- subject – имя субъекта
- открытый ключ KU + параметры

Вся эта информация подписана закрытым ключом сертификационного центра. Проверяющая сторона получает подписанный блок данных, если у нее есть открытый ключ сертификационного центра, то она проверяет всю эту информацию. Каждый из участников может выполняться на какой угодно аппаратной платформе, в какой угодно операционной системе, браузере и прочее.

Когда мы рассматривали алгоритмы создания, проверки подписи, мы, на самом деле, подписывали некое число, то есть нам необходимо все поля сертификата преобразовать в числа, затем получить хэш-код этих данных и подписать их. То есть нам надо иметь возможность единообразно преобразовывать наши разношерстные данные, для этого нам необходим стандарт ASN.1, он используется также при сетевом взаимодействии. На самом деле таких стандартов несколько, которые позволяют нам обмениваться какой-то информацией между разными операционными системами, разными приложениями. Вы наверняка сталкивались с форматами XML, JSON. XML является наиболее популярным форматом, а ASN.1 – это более старые форматы, которые чаще используются в сетях.

Формат представления данных ASN.1

ASN.1 – это С-подобный формат, который очень напоминает написание структур на языке С. Формат XML отличается тем, что он более многословный. Существуют конвертеры, которые могут преобразовывать ASN.1 в XML и наоборот.

Рассмотрим подробнее формат ASN.1, что в нем содержится. В нем есть обычные примитивные типы данных: integer, boolean, string, char, null. Но есть и отличное от языка С, например, чтобы преобразовать поле сертификата «открытый ключ KU + параметры» в число, создается специальный идентификатор OID. Для каждого алгоритма в ASN.1 регистрируется OID (Object IDentificator).

В ASN.1 также есть следующие структуры данных:

- SEQUENCE – упорядоченная последовательность типов данных,
- SEQUENCE OF – упорядоченная последовательность значений,
- SET – не упорядоченная последовательность типов,
- SET OF – не упорядоченная последовательность значений,
- CHOICE – открытый ключ может выбираться из нескольких значений для соответствующих алгоритмов открытого ключа.

Расширения сертификата

Теперь поговорим немного о расширениях сертификата, мы помним, что они появились в третьей версии X.509 v.3. Они так же входят в сертификат и подписываются, как и остальные поля, а это означает, что никто кроме сертификационного центра не сможет изменить их без обнаружения этого проверяющей стороной. Расширение имеет формат:

$$\langle \text{имя} \rangle = \langle \text{значение} \rangle \langle \text{признак критичности} \rangle,$$

где признак критичности принимает значения либо true, либо false. Если расширение критично (true), то проверяющая сторона должна понимать семантику данного расширения, то есть понимать, какие возможные значения могут быть у конкретного расширения, и какое значение является правильным, а какое неправильным для данного сертификата. Если же проверяющая сторона не понимает семантики конкретного расширения, которое положено в сертификат, то она должна считать данный сертификат недействительным. Если расширение не помечено как критичное, и проверяющая сторона не понимает расширения, то решение вопроса о том, признавать ли данный сертификат действительным, остается за проверяющей стороной.

Перечислим основные расширения, которые могут иметь место:

1. KeyUsage – использование ключа.
   Когда мы рассматривали алгоритмы с открытым ключом, мы говорили, что часть из них может использоваться только для создания подписи (DSS), часть – как для создания подписи, так и для шифрования (RSA), то есть возможно разное применение алгоритмов. С точки зрения безопасности считается неправильным, если один и тот же ключ используется и для шифрования, и для подписи. Также считается неправильным – одним и тем же ключом подписывать менее значимые данные, например, частную переписку, и более значимые данные, например, крупную финансовую транзакцию. Расширение KeyUsage перечисляет, для чего может использоваться данный ключ, для которого мы создаем сертификат. Значений у данного расширения может быть достаточно много, например, digital signature, CRL signature, CA only, NonRepOnly, encryption only и так далее.

2. Policy
   Расширение определяет для какой политики создан данный сертификат. Например, у нас есть одна организация CA_1, которая выпускает сертификаты для своих пользователей, в которых указана политика $pol_1, pol_2, \dots$. И пусть есть другая организация CA_2, которая выпускает сертификаты для своих пользователей с политикой $p_1, p_2, \dots$. Наконец, обе эти организации хотят создать между собой защищенный документооборот. В принципе они готовы считать, что политика pol_1 эквивалентна по своим параметрам, например, по длине ключа, политике p_1. Но с точки зрения формальной действительности сертификатов существует большая проблема: формально политики pol_1 и p_1 являются разными, они имеют разные идентификаторы, и без того, чтобы не перевыпустить все сертификаты у каждого из участников, эту проблему не решить. Это большая работа, и получается, невозможно плавно перейти на использование инфраструктуры открытого ключа без снижения безопасности. Соответственно, либо у CA_1, либо у CA_2 создается сертификат, и в нем появляется еще одно расширение policy mapping, в котором указывается, что $pol_1 = p_1$. Таким образом, с помощью расширений мы можем решать проблемы связанные с плавным внедрением безопасных коммуникаций между различными организациями.

3. AltName
   У нас в основном сертификате есть поле имя субъекта subject name, имя сертификационного центра CA name, которые в терминологии LDAP указываются в форме DN. Однако этого не всегда достаточно, у разных субъектов могут быть еще какие-то совершенно другие имена, например, если

субъектом является человек, то хотелось бы ввести поля фамилии, имени, отчества. Соответственно, существует расширение AltName, где уже можно определить все, что угодно, относящееся к данному типу субъекта: для человека – фамилия, для маршрутизатора – его IP-адрес, для сервера – его виртуальный host и так далее.

4. Следующее расширение связано с электронным документооборотом, мы хотим иметь возможность создавать электронные подписи для различных документов, которые можно было бы проверить через достаточно долгий промежуток времени. Но проблема в том, что время жизни ключа ограничено, соответственно, возможна ситуация, когда период действительности закрытого ключа, которым мы подписываем, отличается от периода действительности открытого ключа, которым мы проверяем подпись. Мы уже не можем подписывать этим закрытым ключом новые документы, но проверять существующие документы еще можем, это тоже может указываться в расширении.

5. В расширении можно указать, где брать CRL, например, в формате URL, причем в качестве первой части URL, где указывается протокол, может стоять LDAP, затем в формате DN уже место, где лежит список отмененных сертификатов: ldap:<DN>. Предусмотрена возможность оптимизировать трафик и в URL указывать не полный CRL, а deltaCRL, отражающий, какие изменения произошли относительно предыдущего выпуска.

Способы отмены сертификатов

Вернемся к теме отмены сертификатов, мы уже говорили, что каждый центр выпускает список отмененных сертификатов через определенные промежутки времени ΔT, которые указаны в регламенте данного сертификационного центра. Понятно, что чем ΔT меньше, тем меньше время неопределенности в действительности сертификата, но тем больше нагрузка и на проверяющую сторону, и на сертификационный центр. Это первый способ отмены сертификатов.

Второй способ отмены сертификата OCSP (Online Certificate Status Protocol) достаточно простой. Клиент OCSP обращается к серверу OCSP с вопросом, не отменен ли данный сертификат, а сервер по своим информационным каналам узнает и возвращает ответ: действителен, не действителен или неизвестно. Мы избавились от времени ΔT, в течении которого неизвестен статус сертификата. Плюс использования этого протокола заключается также в том, что клиент может быть маломощным, а

проверка сертификата, как мы помним, это достаточно сложная операция, и связана не только с необходимостью совершения математических операций, но и с необходимостью создания и хранения цепочки сертификатов. Вместо этого у клиента может быть только открытый ключ OCSP сервера, и любую цепочку сертификатов он просто отсылает мощному OCSP серверу, который уже возвращает ответ.

Обновление сертификата сертификационного центра

Понятно, что ключи надо время от времени менять всем, не только пользователям, но и сертификационным центрам. Мы поговорим о том, как это сделать планово, не будем рассматривать экстренные ситуации: если что-то неожиданное и плохое случилось с закрытым ключом сертификационного центра, необходимо сразу же отменять соответствующий сертификат и все сертификаты, выпущенные с использованием данного ключа.

У нас в хранилище должен лежать, так называемый, *самоподписанный сертификат*, то есть сертификат подписан закрытым ключом, а в самом сертификате лежит открытый ключ, соответствующий данному закрытому: $KR \ll KU \gg$. В принципе самоподписанный сертификат к безопасности не имеет никакого отношения, потому что его может создать любой. Единственный его смысл в том, что участник доказывает, что он знает соответствующий закрытый ключ. Когда сертификационный центр считает, что пора выпускать новый сертификат, то есть подходит к концу срок действительности старого сертификата $old \ll old \gg$., он выпускает следующие три сертификата: $new \ll new \gg$, $new \ll old \gg$ и $old \ll new \gg$. Таким образом, он выпускает новый самоподписанный сертификат для новой пары ключей, затем новым закрытым ключом подписывает старый открытый и старым закрытым ключом подписывает новый открытый. Зачем это делается, возможно, что у проверяющей стороны хранится старый открытый ключ сертификационного центра, ей приходит сертификат, подписанный новым ключом, тогда она получает сертификат $old \ll new \gg$ и восстанавливает таким образом цепочку сертификатов. Второй вариант: у проверяющей стороны есть новый открытый ключ сертификационного центра, она получает сертификат, подписанный старым ключом, тогда она получает сертификат $new \ll old \gg$ и так же восстанавливает таким образом цепочку сертификатов. Получается, что в какой-то момент времени в хранилище сертификатов у нас лежат 4 сертификата.

Протокол SSL/TLS

Давайте вспомним сетевые технологии. У нас есть сетевой уровень IP, транспортный уровень TCP и прикладной уровень. Стык протоколов TCP, IP разрабатывался больше 40 лет назад в американских университетах для общения между собой, тогда каналы были плохие, пакеты терялись. Основная задача состояла в том, чтобы создать надежную связь по ненадежным каналам, чтобы пакеты данных доходили, и им это удалось. Однако разработчиков совершенно не заботила безопасность. (Самое интересное передается на прикладном уровне.) Уже в конце 90-х годов было понятно, что необходимо защищать данные, появилось много алгоритмов того же симметричного шифрования. Тем не менее хотелось бы создать некую прослойку между прикладным уровнем и TCP и все, что выше, зашифровать, а также выполнить аутентификацию. Самым распространенным еще в 90-е годы протоколом, который и требовал защиты, был протокол HTTP, и, соответственно, первыми, кто запаниковал по поводу безопасности были браузеры. SSL 3.0 (Secure Sockets Layer) – уровень безопасных сокетов. Вообще говоря, неправильно, если стандарты разрабатывают коммерческие организации, потому что они делают это под себя. Существует организация IETF (Internet Engineering Task Force), которая занимается стандартизацией, она в том числе выпускает rfc. Вскоре после того, как был разработан SSL 3.0, они разработали стандарт, который назвали TLS 1.0 (Transport Layer Security) – безопасность транспортного уровня. Оба стандарта TLS и SSL очень похожи, однако в названиях скорее больше политики, чем здравого смысла.

Какие требования предъявлялись к этим стандартам:

1. Криптографическая безопасность

2. Интероперабильность
   Необходимо создать некий каркас или framework, в который могли бы встраиваться различные протоколы, HTTP-протокол хоть и самый популярный, но не единственный.

3. Расширяемость
   Криптографических алгоритмов у нас много, надо понимать, что мы рассмотрели далеко не все. Хотелось бы, чтобы с разработкой нового алгоритма не пришлось заново разрабатывать или перепрограммировать весь протокол, а можно было бы просто добавить в него новую библиотеку. Прежде чем установить защищенное соединение с использованием того или иного алгоритма участники должны вести переговоры, что бы им использовать, чего не было в протоколе Kerberos.

4. Относительная эффективность

Мы помним, что аутентификация в криптографии с открытым ключом с использованием цифровых подписей, вообще говоря, ресурсоемкая, это всегда возведение в большую степень. Если говорить от протоколе HTTP, как у нас выглядит взаимодействие: клиент делает запрос, используя метод get или post, а сервер возвращает ему html-страницу, после этого TCP соединение завершается. Соответственно, если клиент через какое-то время, посмотрев на html-страницу, которую ему прислали, кликнет и останется на том же самом сайте, на том же самом сервере, не хочется заново начинать аутентификацию, со всеми операциями возведения в степень и так далее. Здесь должно появиться понятие сессии, чтобы не производить одни и те же операции сверх надобности.

Протокол состоит из двух частей: протокола записи и протокола рукопожатия. У нас есть IP, TCP, выше выполняется протокол записи, а еще над ним протокол рукопожатия и прикладной протокол. Протокол записи выступает некой прослойкой сначала для протокола рукопожатия, а затем для прикладного протокола. Основное назначение протокола записи – это конфиденциальность и целостность соединения, причем о ключе договариваются в протоколе рукопожатия. Для конфиденциальности используются алгоритмы симметричного шифрования, для целостности – HMAC, MD5, SHA-1, а то и SHA-2, SHA-3.

Протокол рукопожатия обеспечивает:

1. Переговоры об используемых алгоритмах, тем самым обеспечивается расширяемость.
2. Аутентификация участников, причем клиент устанавливает соединение, но первым себя инициирует сервер.
3. Переговоры об общем секрете, они безопасные, то есть секрет нельзя подсмотреть, и надежные, если выполнена аутентификация сервера.

Теперь поговорим о протоколе записи. У нас на стороне клиента и на стороне сервера есть два состояния: текущее C и ожидаемое S. Каждое состояние описывает алгоритмы симметричного шифрования + ключи, хэш-функции + ключи, которые используются для защиты трафика, обеспечения конфиденциальности и целостности передаваемых данных. Причем алгоритмы и ключи в разных направлениях (клиент-сервер или сервер-клиент) могут быть разными, алгоритмы, правда, скорее всего одинаковые, но ключи точно разные. На самом деле, могут использоваться еще и алгоритмы сжатия. Вначале при установлении соединения у нас все алгоритмы и ключи установлены в null, в протоколе рукопожатия определятся алгоритмы и ключи для ожидаемого состояния, по его завершению ожидаемое состояние становится текущим.

Что касается протокола рукопожатия, инициатором является так же клиент, он посылает серверу ClientHello, то есть наборы алгоритмов: алгоритмы симметричного и

асимметричного шифрования, хэш-функции и алгоритмы сжатия, в каждом из типов алгоритмов они упорядочены по предпочтениям клиента. Посылается также случайное число клиента ClientHello.Random, SessionID и возможно какие-то дополнительные данные, о которых мы поговорим в следующий раз. В ответ сервер посылает ServerHello, где он из каждого набора алгоритмов выбирает один, который у него реализован, также посылает случайное число сервера ServerHello.Random. Об остальном поговорим в следующей лекции.

ЛЕКЦИЯ 12. ПРОТОКОЛ SSL/TLS. ПРОТОКОЛЫ ЗАПИСИ И РУКОПОЖАТИЯ

На прошлой лекции мы рассматривали протокол SSL/TLS, говорили о том, на каком уровне стэка TCP, IP он выполняется. Мы создаем прослойку между протоколом TCP и прикладным протоколом. Все, что будет передаваться по прикладному протоколу, будет обеспечено конфиденциальностью, целостностью, аутентификацией сторон. Протокол SSL/TLS состоит из двух частей: протокола записи и протокола рукопожатия. На стороне клиента и на стороне сервера есть два состояния: текущее C и ожидаемое S. Задача протокола рукопожатия заключается в том, чтобы ожидаемое состояние сделать текущим. Каждое состояние описывает алгоритмы симметричного и асимметричного шифрования, хэш-функции, которые используются для защиты трафика, обеспечения конфиденциальности и целостности передаваемых данных, алгоритмы сжатия и, соответственно, ключи ко всем этим алгоритмам (K, IV, KMAC). Причем ключи в разных направлениях (клиент-сервер или сервер-клиент) могут быть разными.

Протокол рукопожатия

Рассмотрим протокол рукопожатия. Начинает клиент, он посылает серверу:

- ClientHello, то есть наборы алгоритмов, которые у него реализованы: алгоритмы симметричного и асимметричного шифрования, хэш-функции и алгоритмы сжатия, в каждой категории алгоритмы упорядочены по предпочтениям клиента.
- ClientHello.Random – случайное число клиента
- SessionID – идентификатор сессии
- …

  В ответ сервер посылает:

- ServerHello, где он из каждого набора алгоритмов выбирает по одному, который у него реализован.
- ServerHello.Random – случайное число сервера
- Certificate* – сертификат сервера (* означает, что это сообщение может отсутствовать)
- ServerKeyExchange*

Давайте вспомним, как происходит обмен общим секретом, в последних двух сообщениях нам нужно обязательно обменяться общим секретом и, возможно, также сервер должен аутентифицировать себя для клиента. Первый вариант обмена – Диффи-Хеллман, но мы помним, что в этом случае возникнет атака man in the middle. Если мы готовы с этим смириться, и нам важнее эффективность, то можем использовать Диффи-Хеллмана. Тогда ServerKeyExchange – это открытое значение алгоритма, а Certificate может вообще отсутствовать, в этом случае говорят, что у нас идет взаимодействие с анонимным сервером, если же на сервере есть сертификат, то он посылается в виде Certificate. Второй вариант обмена – RSA. Соответственно, можно послать открытый ключ RSA, но лучше бы он, конечно, был подписан, тогда в Certificate посылается сертификат, а сообщение ServerKeyExchange может отсутствовать. Какие могут быть проблемы с сертификатом RSA, в нем может быть указано, что он только для подписи. Мы помним, что алгоритм RSA можно использовать и для подписи, и для шифрования, но нам может не позволить использовать его так, как нам хочется, политика безопасности. В таком случае в Certificate мы посылаем сертификат для одного ключа, а в ServerKeyExchange – либо ключ RSA, либо Диффи-Хеллмана. Далее если сервер себя идентифицировал, то есть сообщение Certificate не пустое, то сервер может запросить сертификат клиента:

- CertificateRequest* (сообщение может и отсутствовать, аутентификация клиента не обязательна)
- ServerHelloDone – это последнее сообщение, оно означает, что серверный блок окончен.

Если у клиента запросили сертификат, он должен его послать:

- Certificate*
- ClientKeyExchange, оно зависит от ServerKeyExchange
- CertificateVerify* (присутствует, только если есть Certificate). Понятно, что, на самом деле, посылка сертификата еще не является аутентификацией, нужна цифровая подпись. Здесь выполняется конкатенация всего того, что было послано и получено клиентом, все это подписывается закрытым ключом клиента, который соответствует тому, что есть в сертификате. Учтем, что если речь идет о первом вызове протокола рукопожатия, то все еще идет в открытом

виде, пока мы никаких секретов не посылаем. Что к данному моменту может сделать нарушитель, он может попытаться изменить наборы алгоритмов в ClientHello, то есть попытаться заставить нас использовать более слабые алгоритмы. Это надо как можно быстрее определить, в CertificateVerify* клиент подписывает все, что он послал и получил.

- ChangeCipherSpec, это сообщение посылают и клиент, и сервер. В этом сообщении опять же выполняется конкатенация всего того, что было послано и получено каждой из сторон, все это шифруется и обеспечивается целостность с использованием тех алгоритмов, о которых они договорились в ClientHello и ServerHello. Даже если не было сообщения CertificateVerify*, то есть клиент не аутентифицирован, то в ChangeCipherSpec участники смогут проконтролировать, не было ли к этому моменту атаки man in the middle.
- Finished, это сообщение также посылают и клиент, и сервер, оно зашифровано новыми алгоритмами и новыми ключами.

Протокол записи

Клиент и сервер вычислили общий секрет определенного размера (pre_master_secret), его длина зависит от того, каким алгоритмом участники вычисляли этот общий секрет, например, от длины ключа RSA или от группы Диффи-Хэллмана. Для обеспечения безопасности участникам нужно 6 ключей:

- client_write_key
- server_write_key
- client_write_IV
- server_write_IV
- client_write_mac
- server_write_mac

Длины этих ключей так же заранее не известны и зависят от того, о чем участники договорились в ClientHello и ServerHello. Таким образом, наша задача – растянуть pre_master_secret, длина которого заранее не известна, на 6 ключей, длина которых заранее не известна. Вообще говоря, мы хотим, чтобы если у нас взломают какой-то один ключ, это не приводило к тому, что у нас автоматически взломаны все остальные ключи. Это свойство называется PFS (Perfect Forward Secrecy). Здесь нам помогают сильные хэш-функции и стандарт HMAC. Хэш-функция называется криптографически сильной, если зная результат хэширования и даже зная какую-то часть сообщения, невозможно восстановить все исходное сообщение. Мы будем хэшировать каждый раз секреты и какие-то еще данные и выполнять конкатенацию, из этого мы нарежем

ключи. Хэшировать будем секрет, всем известное постоянное стандартное значение и то, что в принципе можно подсмотреть, но оно всегда разное, это называется зерном seed, в этом нам помогут ClientHello.Random и ServerHello.Random. У нас есть две хэш-функции:

- HMAC_MD5(secret, data)
- HMAC_SHA1(secret, data)

Определим следующую функцию:

$$P\_hash = HMAC\_hash(secret, A(1)||seed) \,||\, HMAC\_hash(secret, A(2)||seed)|| \dots, \quad (12.1)$$

где $A(1) = seed$, $A(i) = HMAC\_hash(secret, A(i-1)||seed)$.

Осталось определить функцию:

$$PRF(secret, label, seed) = P\_MD5(S_1, label \,||\, seed) \oplus P\_SHA1(S_2, label \,||\, seed). \quad (12.2)$$

Соответственно, мы будем растягивать этот секрет следующим образом: pre_master_secret неизвестной длины растянем на master_secret длиной 48 байт, а его на ключевой материал keys нужной длины. Итак, мы вычислили pre_master_secret, а далее все вычисления каждая сторона (сервер и клиент) делает у себя, больше никто ничего не передает. Было предложение создать эти 6 ключей и зашифровать с помощью pre_master_secret, но этот вариант хуже, чем независимые вычисления.

Мы ввели функцию PRF в виде (12.2), опишем подробнее, из чего она состоит. S_1 – это левая половинка секрета, а S_2, соответственно, правая. Каждая из функций P_MD5 и P_SHA1 выполняется свое определенное количество раз. Хотелось бы, конечно, заменить P_MD5 и P_SHA1 на P_SHA2 и P_SHA3, но проблема в том, что у нас много разных серверов, у которых разная реализация, у них в свою очередь большое количество клиентов, у которых также разная реализация.

Продолжим наши вычисления, находим

$$master\_secret = PRF(pre\_master\_secret, \text{“}master\_secret\text{”}, ClientHello.Random \,||\, ServerHello.Random), \quad (12.3)$$

где “$master\_secret$” – это строка типа string. Далее вычисляем ключевой материал:

$$keys = PRF(master\_secret, \text{“}key\_expansion\text{”}, ClientHello.Random \,||\, ServerHello.Random), \quad (12.4)$$

Однако, при пересылке сообщений что-то может пойти не так, например, не смогли договориться об алгоритмах, не смогли проверить сертификат, что-то неправильно посчитали на одной или на другой стороне. Соответственно, определен протокол Alert или сообщения протокола Alert, которые каждая сторона в любой момент может послать противоположной стороне. В основном это два типа сообщений: warning и error, в случае ошибки соединение прерывается, в случае предупреждения – на усмотрения противоположной стороны.

Сокращенное рукопожатие

Мы с вами рассмотрели протокол рукопожатия, существует также протокол сокращенного рукопожатия. Вспоминаем, что у нас происходит, если говорить от протоколе HTTP: клиент делает запрос, используя метод get или post, а сервер возвращает ему html-страницу, TCP-соединение закрывается. Клиент, получивший html-страницу, скорее всего совершит следующий переход на тот же самый сервер, и что нам снова выполнять полное рукопожатие? Не хочется заново начинать аутентификацию со всеми операциями возведения в степень, понятно, что это все не даром. Соответственно есть вариант сокращенного рукопожатия, когда клиент в $ClientHello$ посылает $SessionID$, которое создается сервером при первом рукопожатии. Сервер смотрит в своем кэше, есть ли у него такой идентификатор сессии, как завершилось предыдущее соединение, если все есть и все хорошо, то сервер посылает сообщение $ServerHelloDone,$ после обе стороны обмениваются $ChangeCipherSpec$ и посылают друг другу $Finished$, который выполняет конкатенацию всего того, что было послано и получено, все это шифруется и подписывается так, как об этом договорились участники. С помощью сокращенного рукопожатия мы увеличиваем эффективность.

Прикладные протоколы семейства TLS

Поговорим о прикладных протоколах, которые использует SSL/TLS. Прикладной протокол должен понимать, что ему нужна безопасность и вначале необходимо выполнить рукопожатие, у него должна быть реализована соответствующая библиотека. Существовали некорректные реализации, когда вначале делается запрос get, сервер понимает, что надо выполнять SSL, инициирует установление соединения, а после этого он должен заставить клиента снова выполнять протокол get.

Разберемся, что у нас происходит с портами. Если мы говорим о протоколе HTTP, то создается новый порт и заново выполняется соединение, а, например, протокол SMTP по какому порту слушал, по такому и продолжает, то есть это не принципиально, переходит протокол на новый порт или нет.

Следующим расширением протокола TLS является, так называемый, расширенный $ClientHello$, то есть клиент может запросить какие-то дополнительные параметры, которые ему необходимы. Если сервер поддерживает эти возможности, то они устанавливают соединение с учетом этих расширенных параметров, если нет, то используется Alert, и, таким образом, соединение не установлено. Итак, что включает в себя расширенный $ClientHello$:

- Имя сервера (в случае протокола HTTP)

- Фрагментация
- Урезанный MAC
- Список корневых сертификатов
- URL OCSP-сервера
- …

Заметим, что некорректно называть это VPN, это классическое клиент-серверное взаимодействие.

Семейство протоколов IPsec

Здесь речь пойдет уже о безопасности на уровне IP, мы хотим опуститься на уровень ниже. Теперь прослойка будет над уровнем IP, мы будем шифровать все, что находится выше в стэке протоколов TCP, IP. Протокол IPsec должен быть реализован на уровне операционной системы, новый тип сокета, через который будет происходить взаимодействие. Также важно отметить, что маршрутизаторы должны поддерживать протокол IPsec, мы не можем отменить всего того, что было сделано, и выкинуть все маршрутизаторы, которые есть в интернете, соответственно, протокол должен быть разработан так, что в случае непонимания маршрутизатором протокола не происходило ничего критического.

Перечислим протоколы:

1. ESP (Encapsulating Security Payload): IP, ESP, UPPER
   Все, что выше в стэке протоколов UPPER будет обеспечено конфиденциальностью, целостностью. Но мы понимаем, что на самом деле выше может быть все, что угодно, и UDP, и TCP, соответственно ESP будет еще и обеспечивать частичную целостность последовательности.

2. AH (Authentication Header): IP, AH, UPPER
   Сейчас этот протокол мало где реализован. Он обеспечивает целостность UPPER-уровня и определенных полей IP-заголовка. Обеспечить целостность всего мы не сможем, но хотелось бы сделать это для IP-адресов.

Перечислим режимы, в которых они могу выполняться:

1. Туннельный: IP_{out}, ESP/AH, IP_{in}, UPPER
   Появляется понятие шлюза безопасности SG (Secure Gateway).

2. Транспортный: IP, ESP/AH, UPPER

ЛЕКЦИЯ 13. СЕМЕЙСТВО ПРОТОКОЛОВ IPSEC. ЧАСТЬ 1

В предыдущей лекции мы начали рассматривать семейство протоколов IPsec. До этого мы рассмотрели безопасность на прикладном уровне и протокол SSL/TLS, это говорит о том, что нарушитель видит, кто с кем общается: ему видны IP-адреса участников и номера их портов. Также приложение должно понимать, что ему нужна безопасность, если же клиент может хотя бы одно сообщение отправить до выполнения протокола рукопожатия, то все плохо.

Соответственно, второй подход к обеспечению безопасности заключается в том, чтобы опустить все эти проблемы ниже в стэке протоколов на уровень IP. Это означает, что реализация будет происходить на уровне операционной системы, приложение уже может ничего не знать. Здесь возникает другая проблема, есть маршрутизаторы, которые не хотят или не имеют возможности понимать наши премудрости.

Существуют следующие протоколы:

1. ESP (Encapsulating Security Payload): IP, ESP, UPPER
   Все, что стоит выше ESP в стэке протоколов (UPPER), он обеспечит конфиденциальностью, целостностью. Но мы понимаем, что на самом деле в UPPER может стоять все, что угодно, в том числе и TCP, соответственно ESP будет обеспечивать также частичную целостность последовательности. Защита целостности, которую осуществляет TCP, это защита от случайного сбоя, если же нарушения преднамеренные, то защиту от них обеспечивает именно ESP.

2. AH (Authentication Header): IP, AH, UPPER
   Сейчас этот протокол не является обязательным и мало где реализован. Он обеспечивает целостность UPPER-уровня и определенных полей IP-заголовка. Обеспечить целостность всего мы не сможем, но хотелось бы сделать это для IP-адресов.

Перечислим режимы, в которых они могу выполняться:

1. Транспортный: $IP, \text{ESP/AH}, \text{UPPER}$
2. Туннельный: $IP_{out}, \text{ESP/AH}, IP_{in}, \text{UPPER}$
   Появляется внешний заголовок IP_{out}. Вводится понятие шлюза безопасности SG (Secure Gateway) – это маршрутизатор, который поддерживает IPsec. Это может быть маршрутизатор, который не умеет делать ничего другого, кроме как переправлять пакеты с одного порта на другой, или операционная система, выполняющая функции маршрутизатора.

Первый вариант использования IPsec – это создание защищенного туннеля от хоста H к шлюзу безопасности SG, где находится локальная сеть (рис. 13.1). Если пакет идет сначала через туннель, затем попадает в маршрутизатор и там идет до какого-то хоста, то, как правило, IPsec выполняется в туннельном режиме. На самом деле, часто производители реализуют на маршрутизаторах как туннельный, так и транспортный режим.

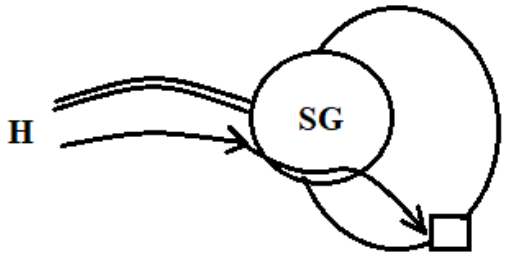

Рис. 13.1. Схема защищенного туннеля.

Второй вариант, имеем два шлюза безопасности, между ними устанавливается защищенный туннель, и еще у каждого своя локальная сеть (рис. 13.2). Типичное соединение двух удаленных офисов. В этом случае тоже может использоваться как туннельный, так и транспортный режим, хотя в стандарте прописан именно туннельный режим.

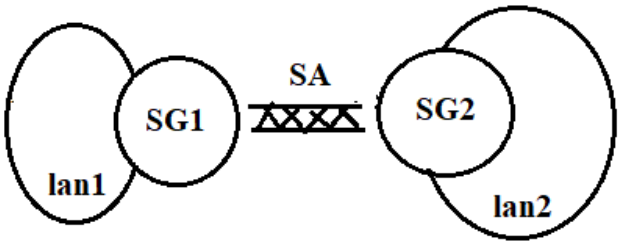

Рис. 13.2. Схема соединения двух шлюзов безопасности защищенным каналом.
Защищенный канал в терминологии IPsec называется SA (Security Association).

В принципе допускается следующее, если у нас какой-то хост H2 тоже поддерживает IPsec, то один канал SA может установиться до SG, а другой до хоста в локальной сети H2 (рис. 13.3). Аналогично и со схемой с двумя локальными сетями: один канал SA установиться между шлюзами безопасности, а другой между хостами H1 и H2 (рис. 13.4).

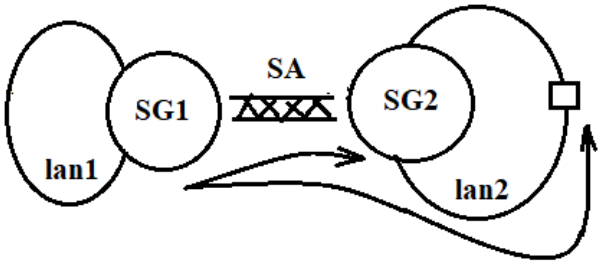

Рис. 13.3. Схема раздвоенного защищенного туннеля.

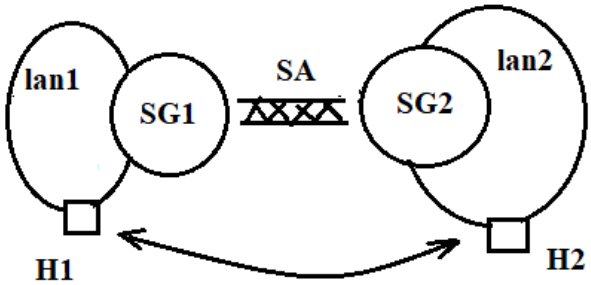

Рис. 13.4. Схема двойного защищенного туннеля между двумя локальными сетями.

Плюс в том, что таким образом мы можем защититься не только от внешних угроз, но и от различных внутренних атак. Конечно, это дается не просто так, по своей топологии эти схемы достаточно сложны. Рассмотрим, что должны иметь шлюз безопасности SG и хост H для того, чтобы поддерживать IPsec. У шлюза есть две базы данных:

SPD (Security Policy Database) – это упорядоченный набор записей, в котором содержится действие Action и параметры трафика. Действия могут быть следующие:

1. Allow – пропустить трафик без обработки
2. Deny – отбросить трафик
3. Обработать с использованием IPsec, то есть зашифровать исходящий трафик и расшифровать входящий.

Каждый входящий и исходящий пакет сравнивается с параметрами трафика и выполняется первое правило, которому соответствует данный пакет. Такая идеология способствует тому, что IPsec достаточно дружественен к другому трафику.

Такой подход, когда есть упорядоченный набор правил rule set, когда у нас последовательно входящий и исходящий трафик сравнивается с набором каких-то правил, и срабатывает первое правило, которому соответствует трафик, это напоминает *межсетевой экран* (Farewall). На самом деле, действий может больше, мы перечислили не все, во-первых, отбросить трафик можно по-разному: молча или послать

отправителю ICMP-сообщение. Во-вторых, есть несколько вариантов, как пропустить трафик. Самый известный способ – это выполнить дополнительно NAT (Network Address Translation). Если маршрутизатор поддерживает NAT, то он меняет IP-адрес исходящего пакета, например, на свой IP-адрес. Фактически SPD – это обычные правила межсетевого экрана, в которые добавлена обработка с использованием IPsec.

Вторая база данных – это SAD (Security Association Database) или база данных установленных ассоциаций, тех которые подняты в соответствии с SPD, и по которым передается трафик. В базе данных указывается следующее, во-первых специфицируется трафик, который следует обрабатывать, затем определяются алгоритмы, использующиеся для шифрования и расшифрования трафика, параметры этих алгоритмов и ключи, время жизни, поскольку ключи надо время от времени менять. SPD создается заранее как политика данного шлюза безопасности или хоста, а SA, которые находятся в базе данных SAD должны устанавливаться. Проблемы возникают те же, что и в протоколе рукопожатия. Во-первых, нам надо договориться об используемых алгоритмах, а их много, и участники могут использовать совершенно разные алгоритмы. Дальше надо аутентифицировать друг друга, хотелось бы иметь варианты аутентификации. Если в протоколах рукопожатия мы могли использовать только цифровую подпись, то здесь хотелось бы использовать более гибкие способы. Если есть возможность использовать общий секрет, то это не запрещено, во многих случаях аутентификация с помощью общего секрета вполне достаточна. Если это не так, то надо использовать криптографию с открытым ключом со всеми вытекающими проблемами: несмотря на то, что это более сильный способ, он же более сложный. Разберемся кого мы будем аутентифицировать, хосты или шлюзы безопасности, это аутентификация на уровне IP-адреса, но этого не всегда достаточно, хотелось ыбы произвести аутентификации. на уровне пользователя, поскольку неизвестно, кто сядет за компьютер.

Протокол IKE

Все вышеперечисленные проблемы решает протокол IKE (Internet Key Exchange). Изначально под IKE подразумевался сам протокол, а вот форматы данных тогда назывались ISAKMP (Internet Security Association and Key Management Protocol), сейчас IETF от него отказались от этого термина, с производителями сложнее, его еще можно найти в логах. В целом IKE определяет и форматы передаваемых данных и последовательность передаваемых сообщений.

Сначала поговорим о форматах данных, вводится понятие SA (Security Association), это SA определяет протокол и набор защиты, то есть то, чем мы будем шифровать данные и чем будем обеспечивать их целостность. Как обычно, одна сторона предлагает несколько наборов защиты, получатель выбирает из того, что ему

предложено то, что у него есть, аналогично это делается в протоколе TLS сообщениями ClientHello и ServerHello. Единственное, мы пока не имеем права использовать терминологию клиент и сервер, потому что мы находимся на сетевом уровне, а не на транспортном, то есть на уровень ниже. Мы будем пользоваться терминами инициатор и респондер. Соответственно, что такое набор защиты: инициатор посылает несколько предложений Proposal, каждое из которых состоит из нескольких преобразований Transform. Преобразованием могут быть DES, AES, SHA-1, SHA-2. Например первое предложение Proposal1 стоит из DES, SHA-1, а второе Proposal2: AES, SHA-2. Респондер выбирает из того, что ему предложили то, что он поддерживает. Мы рассмотрели первый вариант содержимого в протоколах IKE.

Перейдем к рассмотрению следующих типов содержимого:

- KA – это открытые значения Диффи-Хэллмана. Предполагается, что для вычисления общего секрета участники используют алгоритм Диффи-Хеллмана.
- ID – идентификация. Каждый раз мы будем уточнять, на каком уровне происходит идентификация.
- NONCE – случайные числа, как и в протоколе TLS у нас будет возникать проблема, вычислить ключевой материал из общего секрета. Процедура с точностью повторяет TLS.
- HASH с ключом той функции, о которой договорились в Proposal
- Sign – подпись
- CERT – сертификаты
- Notification – уведомления

Протокол IKE выполняется в две фазы. Результатом первой фазы будет создание IKE SA. Результатом второй фазы будет либо создание ESP SA, либо AH SA. Шифрование трафика происходит во второй фазе по безопасным ассоциациям, которые иногда называют дочерними. Вообще говоря, под эгидой IKE SA может быть создано несколько безопасных ассоциаций ESP SA. Далее будем использовать следующее обозначение: у всех обменов используется общий заголовок HDR, если содержимое зашифровано будем ставить звездочку – HDR*.

I. Первая фаза протокола.
    1. Режимы: Main Mode, Aggressive Mode
    2. Способы аутентификации: Pre-shared Secret – разделяемый секрет, цифровая подпись.

Мы можем комбинировать режимы и способы аутентификации. Заметим, что на первых этапах реализации IPsec, в первую очередь в операционных системах, возможно было использовать, так называемый, ручной способ распределения ключей. Это не тот Pre-shared Secret, о котором мы будем говорить, для

которого вычисление ключей алгоритмов шифрования все-таки происходит автоматически.

Main Mode, Pre-shared Secret

У нас есть инициатор I и респондер R, между ними происходит взаимный обмен:

1) $I: HDR, SA \rightleftarrows R: HDR, SA$
   Участники договариваются, что они будут использовать для шифрования и обеспечения целостности.
2) $I: HDR, KE, NONCE \rightleftarrows R: HDR, KE, NONCE$
   Участники обмениваются своими $NONCE$ и открытыми значениями Диффи-Хеллмана KE. Как только I и R обменялись значениями KE, им есть, чем шифровать, поэтому дальше ставим звездочку *.
3) $I: HDR^*, ID, Hash \rightleftarrows R: HDR^*, ID, Hash$
   Происходит идентификация и аутентификация с помощью хэш-функции. Pre-shared Secret нужен только для аутентификации, он не используется в качестве ключа шифрования. Соответственно, выполняется конкатенация всего, что было послано, что было получено, и вычисляется HMAC от Pre-shared Secret.

Main Mode, цифровая подпись

1) $I: HDR, SA \rightleftarrows R: HDR, SA$
2) $I: HDR, KE, NONCE \rightleftarrows R: HDR, KE, NONCE$
3) $I: HDR^*, ID, Cert, Sign \rightleftarrows R: HDR^*, ID, Cert, Sign$

Aggressive Mode, Pre-shared Secret
Если режим Main Mode означает, сделать все получше, то Aggressive Mode означает, сделать все побыстрее.

1) $I: HDR, SA, KE, NONCE, ID \rightleftarrows R: HDR, SA, KE, NONCE, ID$
2) $I: HDR^*, Hash \rightleftarrows R: HDR^*, Hash$
   Здесь в отличие от Main Mode идентификаторы посылаются в открытом виде, там же они зашифрованы. Стоит оговориться, что речь идет об идентификаторах сетевого уровня, в простейшем случае это IP-адреса

Aggressive Mode, цифровая подпись

1) $I: HDR, SA, KE, NONCE, ID, Cert \rightleftarrows R: HDR, SA, KE, NONCE, ID, Cert$

2) $I: HDR^*, Sign \rightleftarrows R: HDR^*, Sign$

Соответственно, им уже есть, чем шифровать, выполнена аутентификация на сетевом уровне, уже вычислен общий секрет с помощью алгоритма Диффи-Хеллмана, они уже могут дальше шифровать свой трафик. Вторую фазу Quick Mode мы будем рассматривать в следующей лекции. Стандартная задача, с которой мы сталкивались и в случае TLS, если у нас что-то случилось с одним ключом, это не значит, что у нас автоматически взломаны все остальные.

ЛЕКЦИЯ 14. СЕМЕЙСТВО ПРОТОКОЛОВ IPSEC. ЧАСТЬ 2

В прошлой лекции мы рассматривали IPsec, два протокола, с помощью которых он защищает трафик: ESP, AH. Мы обсуждали разницу между ними. Ниже ESP в стэке протоколов остается IP-протокол, потому что иначе не смогут работать маршрутизаторы, которые не обязаны понимать IPsec, выше в стэке протоколов стоит UPPER, который мы не будем конкретизировать. У нас есть два режима: туннельный и транспортный. В случае туннельного режима имеем: внешний заголовок IP_{out}, протокол ESP/AH, внутренний заголовок IP_{in} и дальше, как обычно. В транспортном режиме не добавляется внешнего IP-заголовка. Здесь IP_{in} – это IP-адрес внутреннего хоста, который находится за шлюзом безопасности. Мы также рассмотрели две базы данных, которые реализованы на интерфейсе IPsec: SPD и SPA. SPD – это упорядоченная последовательность записей, каждая запись содержит действие, которое необходимо выполнить с пакетом и параметры этого пакета. Это полный аналог правил межсетевого экрана МЭ, выполняется самое первое правило, параметры которого соответствуют параметрам входящего или исходящего пакета. К основным правилам МЭ Allow и Deny добавляется правило – расшифровать входящий пакет или зашифровать исходящий пакет, а параметры и ключи алгоритмов шифрования хранятся в базе данных SAD, в ней хранятся конкретные параметры для данной SA. На прошлой лекции мы также вводили понятие безопасной ассоциации SA – это то соединение, по которому будут передаваться данные. Наконец, прежде чем начать шифровать и обеспечивать целостность данных, то есть прежде чем использовать SA, она должна быть создана. Этим занимается протокол IKE, конечная его цель заключается в том, чтобы создать безопасную ассоциацию SA, которая характеризуется протоколом ESP или AH и адресом получателя. Протокол IKE имеет две фазы: в первой фазе создается ISAKMP SA, результатом второй фазы будет либо создание ESP SA, либо AH SA. Напомню, что в последней версии AH SA не является обязательной. На прошлой лекции мы рассмотрели первую фазу.

I. Первая фаза содержит следующие режимы:
   - Main Mode
   - Aggressive Mode

   и способы аутентификации:
   - Pre-shared Secret
   - Цифровая подпись

   Какой бы мы не использовали режим и способ аутентификации, мы должны договориться о том, чем мы будем шифровать, аутентифицировать противоположную сторону и выработать общий секрет с помощью Диффи-Хеллмана, это передается в содержимом KE.

II. Вторая фаза называется Quick Mode, быстрый режим, в котором нам требуется создать SA уже для целевых протоколов.

Помним, что если у заголовка стоит звездочка HDR^*, значит он будет шифроваться, шифровать будем теми ключами, которые выработали в первой фазе. А вот содержимое SA, в котором передаются предложения о том, какие алгоритмы мы будем использовать для шифрования трафика, идут заново, то есть мы можем изменить алгоритмы, которыми будем шифровать содержательную часть.

Далее у нас идет идентификация отправителя ID_i, идентификация во второй фазе может отличаться от идентификации в первой фазе, где осуществляется идентификация сетевого уровня, в простейшем случае по IP-адресу, хотя возможно производить ее и на транспортном уровне, но не рекомендуется. Также передается $Nonce_i$, он будет использоваться для выработки общего секрета. Дальше может идти новое значение открытого ключа Диффи-Хеллмана, это, так называемое, свойство PFS (Perfect Forward Secrecy) или совершенная безопасность для последующих обменов. На самом деле, мы просто заново обмениваемся открытыми значениями Диффи-Хеллмана, это дает нам то, что ключи , которыми мы будем шифровать основной трафик, абсолютно не зависят от ключей, которые мы использовали на первой фазе. Минус в том, что алгоритм Диффи-Хеллмана несет большую нагрузку на процессор. Но важно, что этот обмен может и не идти.

Вторая фаза хороша тем, что она может повторяться несколько раз. Тогда нарушитель, пытающийся взломать ключ атакой грубой силы, скорее всего не успеет перебрать все значения и найти ключ до перезапуска второй фазы, в которой значение ключа уже поменяется. В принципе ID_i тоже может отсутствовать, если нам хватает идентификации первой фазы, здесь идентификация совершается уже на уровне пользователя.

В итоге обмен выглядит следующим образом:

1) $I: HDR^*, SA, [ID_i], Nonse_i, [KE], Hash_{1i} \rightleftarrows$
   $R: HDR^*, SA, [ID_r], Nonse_r, [KE], Hash_{1r}$

здесь Hash нужен для того, чтобы проверить, что у нас не было атаки man in the middle. Hash получается в результате следующих действий: вычисляется конкатенация всего, что было отправлено и что было получено, к этому присоединяется общий секрет, вычисленный на первой фазе, и считается HMAC с помощью той хэш-функции, о которой договорились участники.

2) $I: HDR^*, Hash_{2i} \rightleftarrows R: HDR^*, Hash_{2r}$

Механизм NAT

Нам осталось рассмотреть NAT (Network Address Translation) – это не механизм защиты, это механизм адресации. У нас есть маршрутизатор NAT, он заменяет адрес источника исходящего пакета в простейшем случае на свой адрес источника, в более сложных конфигурациях – заменяется на какой-то pool адресов. Для возвращаемых пакетов нужно делать обратное преобразование, то есть изменять адрес уже получателя. Вообще говоря, мы разрешаем менять то, что нельзя менять. Вы не найдете ни одной положительной статьи про NAT, тем не менее его используется везде. Мы нарушаем чистоту идеологии стэка протоколов, когда каждый уровень занимается только своими проблемами. Например, протокол FTP имеет проблемы с NAT-ом, чтобы обойти эти проблемы был разработан ALG (Application Layer Gateway), для каждого конкретного протокола он свой. Второй способ заключается в следующем, надо определить:

1) Есть ли поддержка NAT у отправителя и получателя.
   Поддержка NAT означает, что реализация IPsec должна уметь использовать любой порт, на который ему придет пакет, а не только на 500-й, , по умолчанию IPsec использует UDP 500 порт.

2) Есть ли NAT.
   Разберемся, как понять, было ли выполнено между отправителем и получателем преобразование NAT. Каждая сторона знают свои IP-адреса и порты, соответственно, в содержимом первой фазы они передают эту информацию друг другу. Получатель проверяет, если пакет пришел с другого IP-адреса или порта, значит было выполнено преобразование NAT.

3) Добавлено два режима.
   Дополнительно к IP-заголовку добавляется UDP-заголовок для того, чтобы NAT мог преобразовать номер порта, а далее уже идет то, что относится к IPsec: IP, UDP, IPsec. Соответственно, по умолчанию начинает использоваться порт UDPY 500. У нас появились два дополнительных режима UDP.инкапс.транспорт и UDP.инкапс.туннель.

Аутентификация на прикладном уровне

До этого момента у нас аутентификация производилась на сетевом уровне, то есть на уровне IP-адресов, таким образом аутентифицируется компьютер, но не пользователь. Хотелось бы на второй фазе иметь аутентификацию на уровне пользователя. Существуют следующие способы решения этой проблемы:

1) Стандарт XAuth
   Если реализация IPsec поддерживает стандарт XAuth, значит, на второй фазе появляются понятия клиента и сервера. Инициатор предоставляет аутентификатор на прикладном уровне, респондер аутентифицирует. Расположение базы данных, в которой хранятся аутентификаторы зависит от реализации, самые распространенные способы хранения:
   - на SG
   - LDAP/AD (домен)
   - Radius – протокол, разработанный специально для того, чтобы хранить аккаунты отдельно от устройства, которое осуществляет аутентификацию.

На самом деле, еще до того, как появился IPsec, были протоколы, которые кому-то разрешали бы входить в сеть, а кому-то нет. Первым протоколом, который мы рассмотрим, будет PPP (Point-to-Point Protocol) – не маршрутизированный протокол канального уровня, позволял по телефонному соединению выходить в интернет. Аутентификация там производится с помощью протокола CHAP (CHallenge Accept Protocol): у каждого пользователя есть логин и пароль, который хранится на сервере, сервер посылает случайное число Nonce, клиент выполняет хэш-функцию MD5 от пароля и случайного числа и посылает обратно. Это делается для защиты от replay-атаки.

Какое дальнейшее развитие, у нас есть пользователь, который по модему подключается к концентратору доступа через протокол PPP, а концентратор доступа через L2TP-протокол (транспортный протокол второго уровня) соединяется с сервером NAS (Network Access Server), где происходит аутентификация опять же по протоколу CHAP. Заметим, что точка завершения телефонного кабеля и точка завершения PPP не совпадают. Протокол аутентификации CHAP не очень хорош с точки зрения целостности. Как обеспечивается конфиденциальность, участники обмениваются случайными числами, выполняется XOR трафика плюс случайное число. Это очень далеко от шифрования даже с помощью DES, это защищает от случайного просмотра, но если вас захотят взломать, то сделают это легко. Поэтому никто не хочет выходить в интернет с помощью L2TP, но плюс этого протокола в том, что его все поддерживают, он был реализован очень давно.

2) IPsec + L2TP

   IPsec дает нам аутентификации на сетевом уровне, сильное шифрование и целостность, а L2TP – аутентификацию на прикладном уровне.

Протокол GRE

Протокол GRE (Generic Routing Encapsulation) имеет внешний и внутренний протоколы: <out> GRE <in> , где <out> – это транспортная сеть, а <in> – то что нам нужно передать. Очень часто протокол GRE используется, когда внешние и внутренние сети являются IP-сетями: $IP_{out}\ GRE\ IP_{in}$. Протокол нам ничего не обещает ни конфиденциальность, ни целостность, он может лишь обеспечить нам транспорт одной сети по другой сети. Вполне возможно использовать IPsec + GRE, мы шифруем трафик только в том участке, где нам надо, а дальше просто прогоняем его, мы фактически получаем аналог туннельного режима в IPsec.

Рассмотрим последний вопрос, как нам понять, что противоположная сторона не упала и нормально функционирует, а просто отсутствует трафик. Во-первых, можно использовать флажок keep alive, тогда сторона, которая заинтересована в проверке жизнеспособности периодически посылает пакет по данному туннелю. Во-вторых, существует специальный протокол DPD (Dead-Peer-Detection), он более умный и проверяет жизнеспособность, только в отсутствие трафика, а не через определенные промежутки времени.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in

ЛЕКЦИИ УЧЕНЫХ МГУ