ПРЕДИСЛОВИЕ

Это учебное пособие предназначено для широкого круга читателей, знакомящихся с криптографией и применением её в реальной жизни. Пособие построено таким образом, чтобы даже неподготовленный читатель в первой части получил достаточные для усвоения материала теоретические знания. Во второй части собраны описания и реализации реальных наиболее распространённых алгоритмов и протоколов. В третьей части приводится описание технологии IPSec с примером настройки защищённого шифрованного туннеля (IPSec) для маршрутизаторов Cisco. Пособие снабжено несколькими тренировочными заданиями, ответы на которые приводятся в Приложении.

Данное учебное пособие не претендует на научно-исследовательский статус, но автор надеется, что оно поможет более глубокому и всестороннему освоению основ практической криптографии.

ВВЕДЕНИЕ

Прежде чем начать, хочется акцентировать внимание читателя на следующих моментах:

1. Все приведённые здесь данные не являются конфиденциальными и доступны для углублённого изучения в открытых источниках
2. Все протоколы, входящие в группу с общим названием IPSec стандартизированы, поэтому настройка IPSec возможна на любом оборудовании, поддерживающем этот стандарт, а не только на маршрутизаторах Cisco.
3. В теоретической части данного пособия рассматриваются лишь весьма простые положения, опирающиеся на некоторые гипотезы и теоремы, рассмотрение и доказательство которых выходит за рамки данного пособия

ОГЛАВЛЕНИЕ

ЧАСТЬ I. ТЕОРЕТИЧЕСКАЯ БАЗА. АЛГОРИТМЫ.

Глава 1. Введение.

Задача сокрытия информации от нежелательных глаз возникла, наверно, вместе с человечеством. С появлением письменности и грамотности методы сокрытия (шифры) становились всё изощрённее и изощрённее. До нас дошли описания некоторых способов шифрования (алгоритмы), позволяющие передавать информацию через ненадёжные руки. Но мало было сокрыть информацию, её надо было уметь «раскрыть», иначе терялся смысл такого «шифрования».

На современном языке мы бы сказали так: для создания алгоритма шифрования необходимы:

1. Механизм шифрования и параметры, задающие уникальность шифра
2. Механизм дешифрования (очевидно, что он должен зависеть от тех же параметров, что и механизм шифрования)

История знает много самых разных - от бесхитростных до весьма изощрённых – алгоритмов шифрования. Например, со времён войны в Спарте (V в до н.э.) известен шифр «Сцитала». Для его реализации брался жезл определённой толщины, на него наматывалась тонкая лента папируса (без нахлёстов и пробелов), потом на папирус вертикально наносили текст. Размотанная лента представляла для непосвящённых лишь хаотический набор букв. (Такие шифры, где шифрованный текст представляет собой перестановку букв реального текста, называются перестановочными). Или ещё пример: шифр Цезаря. Для его реализации вместо каждой буквы реального текста писалась третья по счёту за ней (алфавит при этом считался «закольцованным», т.е. в русском алфавите после «Я» шла бы «А»). (Такие шифры, где множество букв реального текста заменяется на некое другое множество, называются шифрами замены). Легко видеть, что в случае с шифром «Сцитала» параметром, влияющим на уникальность шифра, является толщина жезла (а если эту толщину ещё и переменной сделать?), а в шифре Цезаря – величина сдвига. Надо отметить, что эти алгоритмы являются весьма слабыми с точки зрения грамотного взломщика. В качестве разминки посчитайте:

Задача 1:
У вас в руках лента с буквами. Все буквы вам известны. Вы грамотны и знаете алгоритм шифрования (т.е. знаете, что для шифрования брался жезл неизвестной для вас толщины и наматывался папирус). Как вы будете взламывать шифр? Сколько попыток (атак на шифр) Вам точно хватит, если предположить, что всего на папирусе N символов, включая пробелы, и в вашем языке «очень мало» однобуквенных слов? Ответ Вы найдёте в конце этого пособия.

Таким образом, против грамотного взломщика эти шифры не устоят. Создатели таких шифров, надеялись, что сам процесс шифрования взломщику неизвестен. Однако, было бы наивно полагать, что когда-нибудь этот процесс не станет достоянием общественности. Поэтому возникает, во-первых, понятие стойкости алгоритма, а во-вторых, градация атакующих по уровню осведомлённости. Ясно, что если алгоритм стоек против самого сильного и осведомлённого противника, то он стоек как минимум на столько же против любого более слабого атакующего. Давайте опишем его, этого самого опасного и сильного атакующего.

- Атакующий может перехватывать любые пакеты, проходящие через компьютерную сеть, анализировать их и даже подделывать (активный атакующий)
- Атакующий может делать это незаметно для обменивающихся сторон. Этот момент очень важен, ведь если атакующего легко обнаружить, значит можно принять некоторые меры.
- Атакующий знает механизм шифрования, однако он не знает параметра, от которого зависит алгоритм шифрования, или, другими словами, он не знает ключа. Для некоторых алгоритмов для шифрования и дешифрования используются разные ключи (система с открытым ключом). В этом случае предполагается, что атакующий знает открытый ключ и не знает закрытый.
- Атакующий обладает адекватной технической базой и вычислительными мощностями

Задача криптографии состоит в том, чтобы создавать алгоритмы, стойкие именно против такого, сильного противника. Причём в идеале нельзя допустить, чтобы атакующий получил даже один бит исходной информации со 100% вероятностью, не говоря уже обо всём тексте.

Приступая к более детальному рассмотрению задач криптографии, давайте формализуем стоящую перед нами задачу:

1. Мы разрабатываем стойкие алгоритмы против самого сильного противника
2. Предполагается в некоторых случаях, что есть способ связи внесетевой, безопасный для одной транзакции, т.е. мы можем передать (курьером, голубями) один раз ключ для шифрования или аутентификации (подтверждения подлинности источника и получателя)
3. Не существует абсолютно стойкого алгоритма[1], т.е. не взламываемого ни при каких условиях, ибо всегда есть метод полного перебора. Он требует огромного количества ресурсов, но всегда даёт результат. Поэтому можно говорить лишь об алгоритмах, не имеющих эффективных (быстрых) способов перебора всех вариантов ключей. Т.е. чтобы подбор ключа для атакующего был сложной задачей. Под сложностью понимается количество вычислительных ресурсов, необходимых для взлома шифра методом полного перебора. Будем говорить, что алгоритм эффективен, если шифрование и дешифрование с известными ключами выполняется достаточно быстро, а дешифрование с неизвестным ключом требует времени, по истечении которого актуальность информации теряется.

[1] Вообще говоря, абсолютно стойкий шифр, есть. Как доказал Шеннон, существует единственный абсолютно стойкий шифр, ключом для которого является случайная последовательность, которая может использоваться лишь один раз. При этом осуществляется побитовое сложение *n* бит ключа с *n* битами текста. Однако, такой алгоритм очень трудоёмкий из-за необходимости генерировать и хранить большой объём случайных ключей.

Глава 2. Группы, кольца, поля.

Для последующего изложения потребуются некоторые элементарные знания из области вычислительной математики, теории чисел и теории сложности. Приведем здесь без доказательств некоторые факты из этих областей.

Определение: Множество элементов G называется группой с операцией "$\circ$", если выполняются следующие условия:

1. Ассоциативность. Для любых трёх элементов порядок выполнения операции не важен $\forall\, a,b,c \in G \quad (a \circ b) \circ c = a \circ (b \circ c)$

2. Существует элемент e из G называемый единицей группы, такой что $\forall\, a \in G \quad a \circ e = e \circ a = a$

3. Для любого элемента множества существуют левый и правый обратные элементы, принадлежащие этой группе $\forall\, a \in G\, \exists\, b_1, b_2 \in G : a \circ b_1 = b_2 \circ a = e$

Из этого определения следует несколько простых следствий:

- Порядок выполнения операции не важен не только для трёх, но и для любого количества элементов. Поэтому скобки, как правило, опускают.
- Единица единственна в группе. Действительно пусть есть две единицы $\exists\, e_1, e_2 \in G \Rightarrow\ e_1 = e_1 \circ e_2 = e_2$

Если к требованиям общего определения добавить требование коммутативности:

4. $\forall\, a,b \in G : a \circ b = b \circ a$

то мы получим так называемую абелеву группу.

Для абелевой группы легко видеть, что левый обратный равен правому обратному:

$\forall\, a \in G\, \exists\, b_1, b_2 \Rightarrow\ a \circ b_1 = b_2 \circ a = a \circ b_2 \Rightarrow\ (b_2 \circ a) \circ b_1 = (b_2 \circ a) \circ b_2 \Rightarrow\ b_1 = b_2$

В дальнейшем изложении мы будем использовать лишь абелевы (коммутативные) группы.

В качестве иллюстрации приведу несколько примеров различных групп

1. Рассмотрим множество матриц размера *n*x*n* со стандартной операцией умножения матриц. Тогда такое множество является группой. Причём некоммутативной (т.е. не абелевой). Единицей такой группы является матрица, у которой на главной диагонали стоят все единицы, а на всех остальных местах – нули.
2. Множество действительных чисел без нуля с обычной операцией умножения является группой, причём абелевой.

Определение: группа, образованная всеми степенями некоторого элемента *g*, называется циклической подгруппой, порождённой *g* и обозначается <*g*>

Поясню: Пусть есть группа G с операцией умножения. Тогда в этой группе можно ввести операцию возведения в целую степень:

$g \in G$

$g^m = \underbrace{g * \ldots * g}_{m\ раз} \in G$

$g^0 = e \in G$

$g * g^{-1} = e$

$g^{-m} = \underbrace{g^{-1} * \ldots * g^{-1}}_{m\ раз} \in G$

Если в такой подгруппе все элементы различны, то она бесконечна. Рассмотрим вариант, когда есть такие k и l, что $g^k = g^l \Rightarrow g^{k-l} = e$, т.е. группа конечна.

Определение: минимальное такое m, что $g^m = e$ называется порядком элемента g в циклической группе G

Определение: Кольцом называется множество K с операциями сложения и умножения, обладающее следующими свойствами:

1. Относительно сложения K есть абелева группа (её ещё называют аддитивной группой кольца K)
2. $\forall a,b,c \in K \Rightarrow a(b+c) = ab + ac, \quad (a+b)c = ac + bc$ - дистрибутивность умножения относительно сложения.

Определение: кольцо K называется коммутативным, если $\forall a,b \in K \Rightarrow ab = ba$

Определение: полем называется коммутативное, ассоциативное кольцо с единицей, в котором всякий ненулевой элемент имеет обратный, принадлежащий этому кольцу.

Примерами полей служат хорошо известные множества действительных чисел и комплексных чисел с обычными операциями сложения и умножения.

Глава 3. Кольцо вычетов. Функция Эйлера.

Рассмотрим далее конструкцию, называемую кольцом вычетов, ибо она нам понадобится для дальнейшего изложения.

Определение: Вычетом r числа a по модулю n называется остаток от деления a на n и записывается так:

$r = a \bmod n$

Совокупность всех остатков от деления на n образуют множество G

$G_n = \{\forall r : 0 \le r < n\}$

Определение: Мощностью множества называется количество элементов этого множества и обозначается $|G|$

Очевидно, что $|G|=n$

Если в этом множестве ввести операции сложения и умножения по модулю n получится кольцо вычетов по модулю n, которое часто обозначают Z_n:

$\forall a,b \in G_n$

$\exists c = (a+b) \bmod n \in G_n$

$\exists d = (a*b) \bmod n \in G_n$

Элементы c и d называются соответственно суммой и произведением по модулю n.

Утверждение: кольцо вычетов является полем тогда и только тогда, когда число n – простое.

В случае, когда Z_n - поле, из него можно выделить **мультипликативную группу поля**: это все элементы кольца без нуля с операцией умножения. Будем обозначать такую группу $Z_n^* = \{Z_n \setminus 0, *\}$

Определение: Мультипликативная группа *G* называется циклической, если существует такой элемент $g \in G$, что $G = < g >$. Всякий такой элемент называется порождающим.
Утверждение: Любой элемент мультипликативной группы поля вычетов (кольца с простым модулем) является порождающим.

Обозначим как *(a,b)* – максимальный общий делитель натуральных чисел *a* и *b.*
Если *(a,b)*=1 то *a* и *b* называют взаимно простыми.

Введём функцию Эйлера от натурального *n* как количество натуральных чисел, взаимно простых с *n* :
$\varphi(n) = k, k = |P|, P = \{m \in N : (m,n) = 1\}$
Теорема (Малая теорема Ферма, в частном случае ещё называют Теоремой Эйлера):
$\forall g \in Z_n : (g,n) = 1 \Rightarrow g^{\varphi(n)} \bmod n \equiv 1$
Эта теорема нам понадобится при рассмотрении алгоритма RSA.

Напомню, что разложением натурального числа на простые сомножители называется:
$\forall n \in N \exists! p_1,...,p_k,l_1,...,l_k : n = p_1^{l_1} * ... * p_k^{l_k}, все\ p - простые, \forall i < j \Rightarrow p_i < p_j$
Отсюда легко видеть, что
$\max_i p_i \le \sqrt{n}$
Этот простой факт нам пригодится при оценке сложности алгоритмов разложения на простые сомножители.
Функция Эйлера обладает некоторыми замечательными свойствами.
1. Очевидно, что значение функции Эйлера легко вычислить для простого числа, а именно:
$\forall n \in N, простое \Rightarrow \varphi(n) = n - 1$
2. Для любых двух взаимно простых чисел верно следующее:
$\forall k,m \in N : (k,m) = 1 \Rightarrow \varphi(k * m) = \varphi(k) * \varphi(m)$
3. Для любого числа, которое является степенью простого числа, верно следующее:
$\forall p \in N - простое, \forall k \in N \Rightarrow \varphi(p^k) = p^{k-1} * (p - 1)$
Пользуясь этими свойствами легко получить значение функции Эйлера для любого натурального числа, зная его разложение на простые множители.
$n = p_1^{l_1} * ... * p_k^{l_k} \Rightarrow \varphi(n) = \varphi(p_1^{l_1}) * ... * \varphi(p_k^{l_k}) = p_1^{l_1-1} * ... * p_k^{l_k-1} * (p_1 - 1) * ... * (p_k - 1)$

***Задача 2**: Посчитайте, для тренировки, следующие значения φ(30), φ(1024), φ(200560490130)*

Глава 4. «Сложные задачи». Критерии оценки сложности.

Давайте вместе порассуждаем, что можно назвать сложным в вычислительной математике. Из общих соображений понятно, что тот объём вычислений, который был доступен, скажем, 10 лет назад лишь на сверхкомпьютерах, теперь выполняется любой персоналкой раз в 10-100 быстрее и это далеко не предел. Поэтому нам нужен некий механизм оценки сложности алгоритмов, который не зависел бы от конкретного вычислительного аппарата, но мог бы отделить допустимый уровень вычислений от практически нереализуемого, т.е. неэффективного.

Строгая формализация задач по теории сложности требует введения понятия машины Тьюринга – это совокупность алфавита, запоминающего устройства (бесконечная лента с ячейками) и выполняемых операций. Мы же попробуем обойтись без неё, дабы не перегружать это пособие. Попробуем обойтись общими соображениями.

Понятие **алгоритма** вообще описывает:
1. Выполняемые действия
2. Последовательность действий
3. Критерий остановки алгоритма.

В теоретической криптографии принято работать с двоичными строками длины n. Вообще говоря, n может пробегать все натуральные числа, однако применительно к практике обычно выбирают достаточно удобный модуль (блок, длину строки). Скажем, типичным является выбор n=1024 бита. Такой блок удобно размещать в памяти, удобно обрабатывать и т.д. Длину n можно считать параметром безопасности, ведь чем больше n, тем сложнее (и дольше) атаковать схему шифрования.
Пусть есть алгоритм шифрования E и алгоритм дешифрования D, зависящие от ключей шифрования K_1 и дешифрования K_2 соответственно. На вход им подаётся некая строка длины n, а на выходе получаем зашифрованную строку длины n (вообще говоря, длина выходной строки может быть и больше, однако для простоты будем предполагать, что выходная строка равна по длине входной). Тогда верно следующее для любой двоичной строки длины n:

$$\forall x \in \Sigma^n, \Sigma = \{0,1\} \Rightarrow D_{K_2}(E_{K_1}(x)) = x$$

Понятно, что для практики необходимо, чтобы и алгоритм шифрования E, и алгоритм дешифрования D были бы достаточно просты (эффективны) при известных ключах, т.е. чтобы законным пользователям было бы просто их использовать. Желательно также, чтобы алгоритм дешифрования был бы максимально сложен при неизвестном ключе, чтобы максимально усложнить жизнь взломщикам шифров. Вопрос о существовании таких алгоритмов (с использованием соответствующих функций) мы обсудим чуть ниже в этой главе. Пока же, вслед за Эдмонсом, примем следующий

Тезис: Алгоритм считается эффективным, если время его выполнения ограничено полиномом (многочленом) от длины входного слова.

Попробуем разобраться, что значит «время ограничено полиномом»
Любой алгоритм, как вы уже знаете, содержит описание некоторых действий, а критерий для остановки позволяет оценить количество действий, необходимых для успешного окончания алгоритма. Применительно к нашей теме, все операции можно свести к сложению, вычитанию, умножению, делению и возведению в степень n-значных двоичных чисел. Сложность этих операций можно оценить в количестве битовых операций.
Например, на сложение двух n-значных двоичных чисел требуется:
1. n операций собственно сложения
2. не более чем n запоминаний бита (вспомните сложение в столбик).

Помещение переносимого бита (запоминание бита) тоже будем считать за операцию, ибо на это требуется время. Можно считать, что на одну операцию первого и второго типа требуется время *t-const*. Конечно, для разных компьютеров, t – разное, но на эффективность алгоритма в принятых нами предположениях это не влияет.
Таким образом, на сложение двух n-значных чисел требуется времени не более

$$(n + n) * t = 2tn$$

Для упрощения последующего изложения напомню, что такое О-большое и о-малое. Эти общепринятые термины удобны для описания взаимоотношений между функциями: кто растёт быстрее и на сколько. Т.е. с помощью этого языка легко описать асимптотическое поведение функции (т.е. поведение функции при приближении аргумента к предельному значению, например к бесконечности).

Для исследования различных свойств функций часто применяют следующие записи

$$f(x) = \underline{\underline{O}}(g(x)) \Leftrightarrow \exists C - const : \lim_{x \to \infty} \frac{f(x)}{g(x)} = C$$

$$f(x) = \overline{\overline{o}}(g(x)) \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

Читается так: функция *f(x)*- «о-большое (о-малое)» по отношению к функции *g(x)*
В этих обозначениях требуемое количество операций на сложение двух n-значных чисел
$2nt = \underline{\underline{O}}(n)$
Аналогична оценка количества операций для вычитания. Более интересны результаты для умножения и деления: существуют эффективные алгоритмы, использующие $\underline{\underline{O}}(n \ln n \ln \ln n)$ битовых операций. Напомню известные соотношения:

$\forall \alpha > 0 \ln n = \overline{\overline{o}}(n^{\alpha})$

$\forall C > 0 - const : n^{\alpha} = \overline{\overline{o}}(e^{C * n})$

$e^{C * n} = \overline{\overline{o}}(n^{n})$

Легко видеть, что время выполнения алгоритмов умножения и деления ограничено полиномом (многочленом) от *n*:

$$f(n) = \underline{\underline{O}}(n \ln n \ln \ln n) = \overline{\overline{o}}(n^2)$$

Алгоритм возведения в степень по модулю *n* требует и того меньше, а именно $\underline{\underline{O}}(\ln n) = \overline{\overline{o}}(n)$

После этих простых примеров легко понять, что такое полиномиальный алгоритм.
Полиномиальным называется алгоритм, если существует константа *k* и полином степени не выше *k* ($P_k(n) = n^k + c_{k-1} * n^{k-1} + c_{k-2} * n^{k-2} + ... + c_0, c_i - const$) такой, что количество битовых операций, необходимых для выполнения этого алгоритма оценивается как $\underline{\underline{O}}(P_k(n))$
Теперь сформулируем задачу криптографии строго:
Схема шифрования, содержащая алгоритм шифрования *E*, зависящий от ключа шифрования K_1 , алгоритм дешифрования *D*, зависящий от ключа дешифрования K_2 эффективна если:

1. Существует полиномиальный алгоритм, вычисляющий $\forall x \in \Sigma^n \; E_{K_1}(x)$
2. Существует полиномиальный алгоритм, вычисляющий $\forall x \in \Sigma^n \; D_{K_2}(x)$
3. Не существует полиномиального алгоритма, вычисляющего с вероятностью 100% $\forall x \in \Sigma^n \; E_{K_1}^{-1}(x)$, т.е. вычисляющего «обратный» алгоритм к алгоритму шифрования.

Напомню, что мы строим эффективную схему шифрования, в расчёте на самого сильного атакующего, а он знает и алгоритм *E*, и алгоритм *D*, и ключ для шифрования K_1 (это открытый, т.е. общедоступный ключ), может перехватывать любые сообщения. Не знает лишь одного – ключа для дешифрования K_2 (закрытый ключ).
Рассмотрим простой в реализации алгоритм полного перебора. В худшем случае для его выполнения требуется $\underline{\underline{O}}(2^n * T(n))$ элементарных битовых операций, где *T(n)* –

полиномиальное время, необходимое для выполнения одного шага шифрования по известному $E_{K_1}()$ и сравнения полученного значения с перехваченным. Видно, что этот алгоритм не полиномиальный.
Требованием 100% вероятности в третьем пункте нельзя пренебречь, из-за наличия весьма простого в реализации алгоритма угадывания. Однако срабатывает этот алгоритм с пренебрежимо малой вероятностью – таковой принято считать вероятность, меньшую чем $1/P(n)$ для любого полинома *P(n)* при достаточно большом параметре безопасности n .

Теперь, ознакомившись с азами теории сложности, применительно к криптографии, рассмотрим основные направления исследований в этой области и некоторые результаты.

Глава 5. Основы современной теоретической криптографии. Односторонние функции. Применение «сложных задач».

В 1976 году молодые американские математики Диффи и Хеллман опубликовали работу под названием «Новые направления в криптографии», поставив в ней несколько новых интересных задач, не решённых до сих пор. Эта работа всколыхнула интерес к теоретической криптографии и выявила новый подход к разработке практических реализаций различных криптографических протоколов. Давайте кратко ознакомимся с этой работой и некоторыми результатами.

Односторонние функции.

Центральным понятием современной теоретической криптографии является понятие односторонней функции. Строгое определение здесь приводить не буду в силу его громоздкости. По сути, функция является односторонней, если существует эффективный (полиномиальный) алгоритм вычисления значения этой функции от любого аргумента и не существует эффективного (полиномиального) алгоритма инвертирования этой функции (т.е. нахождения любого прообраза по заданному значению функции). Существование таких функций – не доказано, но и не опровергнуто, однако есть веские основания полагать, что такие функции существуют.
Самой распространённой функцией, которую считают односторонней, является $y = \alpha^x \bmod n$, а задача поиска x по заданным y,n,α называется задачей дискретного логарифмирования. На данный момент неизвестно эффективного (полиномиального) алгоритма дискретного логарифмирования.

Ещё одной сложной задачей математики является разложение числа на простые сомножители. Казалось бы, задача проста: берем число и делим его с остатком последовательно на разные простые числа, коих уже известно немало. Алгоритм деления с остатком – полиномиален. Вроде бы алгоритм разложения на простые сомножители логически прост. Давайте оценим вычислительную сложность.

Из аналитической теории чисел известна нижняя асимптотическая оценка количества простых чисел на отрезке $[1,n]$ для достаточно больших n. Приведем её здесь без доказательства: $\frac{n}{\ln n}$

Таким образом, сам алгоритм деления с остатком полиномиален от n, но выполнить его надо не менее чем $\frac{2\sqrt{n}}{\ln n}$ раз для 100% вероятности успеха, ибо мы будем производить поиск простых делителей на отрезке $[1,\sqrt{n}]$. Тогда, для нахождения разложения числа, записываемого 100 десятичными цифрами, придётся перебрать не менее $\alpha \in [0.1;1] \Rightarrow \frac{2\sqrt{\alpha * 10^{100}}}{\ln \alpha * 10^{100}} = \frac{2 * 10^{50}\sqrt{\alpha}}{\ln \alpha + 100\ln 10} > \frac{0.66 * 10^{50}}{-2.3 + 100 * 2.3} = \frac{660 * 10^{47}}{227.7} > 2 * 10^{47}$ чисел!
Приблизительные подсчёты показывают, что компьютеру, производящему 1000000000 (миллиард!) делений в секунду, может потребоваться не меньше 10^{32} лет (!) для разложения такого числа на простые сомножители. Известны и более быстрые алгоритмы разложения на простые множители, но и они работают слишком медленно.

***Задача 3**: Посчитать асимптотическую оценку количества простых чисел, которые надо перебрать, чтобы разложить на простые сомножители числа, длиной 512, 768, 1024 и 2048 бита. Длины чисел выбраны не с проста: такие длины ключей (читай, натуральных чисел), использует алгоритм RSA (512 по умолчанию) и алгоритм Диффи-Хеллмана (соответственно, группа 1, 2 и 5).*

Глава 6. Хэш-функции.

При передаче информации возникает задача проверить, не были ли данные изменены в процессе передачи (случайно или умышленно). Поэтому потребовался способ сопоставлять любому сообщению некоторое число, причём:

1. Разным сообщениям должны соответствовать разные числа
2. Алгоритм вычисления значения для любого сообщения должен быть эффективным (полиномиальным)
3. Алгоритм поиска сообщения по сопоставленному числу должен быть максимально сложным (неполиномиальным)
4. Из соображений удобства, все сопоставляемые числа должны быть одинаковой длины

Для решения этой задачи были придуманы так называемые хэш-функции (от англ. Hash – мешанина). Они разным сообщениям сопоставляют разные значения (хэши) одинаковой длины. Причём неизвестно эффективного алгоритма, который бы находил само сообщение по его хэшу (собственно, если сообщение длиннее, чем сопоставляемый ему хэш, то такой алгоритм вообще не может существовать). Принцип работы этих функций полностью соответствует их названию: они поблочно «перемешивают» исходное сообщение, добавляя в процессе несколько ключей (констант).

Рассмотрим в качестве примера часто используемый и принятый в качестве стандарта в США алгоритм SHA (Secure Hash Algorithm).
В начале своей работы этот алгоритм разбивает исходное сообщение на блоки по 512 бит. Если само сообщение не кратно 512 битам, то оно добавляется справа единицей и последовательностью нулей до кратного 512 битам значения. В конец последнего блока дописывается код длины сообщения, в результате сообщение приобретает вид n двоичных слов. Алгоритм хэширования использует 5 функций и 5 констант для преобразования. В основу алгоритма положены различные логические (булевы) функции, которые побитово преобразуют исходный текст и используемые константы. В итоге получается 5 чисел длиной

32 бита, которые составляют так называемый дайджест (digest) сообщения общей длиной 160 бит.

Алгоритм MD5 (Message Digest 5) также использует различные логические функции (также 5 разных) и несколько констант. На выходе получается дайджест длиной 128 бит, поэтому считается, что MD5 менее стоек, чем SHA, однако на практике ни тот ни другой взломаны не были.

При использовании хэш-функций возникает вопрос: коль скоро любому сообщению ставится в соответствие хэш фиксированной длины, следовательно могут существовать (и существуют!) различные сообщения, которые имеют одинаковый хэш. Такая ситуация называется коллизией. Вопрос нахождения коллизий – нерешённый вопрос, однако известно, что вероятность образования коллизий в реальной жизни пренебрежимо мала. Однако, приведу пример, когда наличие такой коллизии может здорово повредить. (По непроверенным публикациям есть информация, что в Китае был найден эффективный алгоритм поиска так называемых коллизий алгоритма MD5. Дуя на воду лучше использовать SHA)

Предположим, что есть Алиса, которая хочет подготовить два договора: один выгодный для Боба, второй – разорительный для него. Составив такие два договора, Алиса вычисляет хэш по обоим. Потом, путём малых изменений одного и другого, пытается найти пару договоров таких, чтобы их хэши совпадали. Одним из замечательных результатов математической статистики является тот факт, что вероятность обнаружить такую пару за разумное время (а в нашей задаче, вообще говоря, у Алисы много времени) достаточно велика. Поэтому у Алисы есть шанс изготовить 2 договора, выгодный договор предъявить Бобу, а невыгодный – некоторому арбитру (суду) тогда, когда Боб подаст в суд на Алису за нарушение условий договора. Тогда Боб с удивлением обнаружит, что именно «этот», разорительный, договор он подписывал.

ЧАСТЬ II. РЕАЛИЗАЦИИ АЛГОРИТМОВ И ПРОТОКОЛОВ

Глава 1. Алгоритм RSA (Rivest-Shamir-Adleman)

Распространённый алгоритм асимметричного шифрования (т.е. когда для шифрования и дешифрования используются разные ключи). Пожалуй, на сегодняшний момент самая удачная и эффективная реализация идеи с распределёнными ключами.
Алгоритм RSA устроен следующим образом:

Выбираются два достаточно больших, простых и приблизительно равных по длине числа (длиной числа можно считать количество знаков, которыми число записывается в той или иной системе счисления. Обычно, берётся десятичная запись). В качестве модуля берется произведение этих простых чисел.
$p, q - простые, m = p * q$
Выбирается ключ для шифрования *e* так, чтобы он был взаимно прост с *(p-1)* и *(q-1)*.
$e : (e, p - 1) = (e, q - 1) = 1$

$\varphi(m) = \varphi(p * q) = (p - 1) * (q - 1) \Rightarrow (e, \varphi(m)) = 1 \Rightarrow$

$\exists! d : e * d \bmod \varphi(m) = 1$

Тогда ключ (его называют открытым) *e* взаимно прост с *φ(m) (напомню, что это – функция Эйлера)* и существует, причём единственный, ключ для дешифрования *d* (его, как вы уже догадались, называют закрытым)
При этих предположениях, верно следующее:
$\forall x < m \quad y = x^e \bmod m$

$\exists k \geq 0, целое$

$y^d \bmod m = (x^e \bmod m)^d = x^{e*d} \bmod m = x^{1+k*\varphi(m)} \bmod m = (x \bmod m) * (x^{\varphi(m)} \bmod m)^k = x \bmod m = x$

Т.е. получатель берет присланное зашифрованное сообщение, возводит его в степень своего закрытого ключа, берет модуль (остаток от деления на *m*) и получает исходное сообщение.
Ясно, что алгоритмы шифрования и дешифрования с известными ключами – полиномиальны. Мало того, они ещё и очень быстры, ибо требуемое количество операций оценивается, как $\underline{\underline{O}}(\ln m)$. Однако, для взломщика, не знающего закрытого ключа, требуется разложить на множители число *m*, что и является (см. Часть I, Глава 2) основной сложностью.

Глава 2. Алгоритм Диффи-Хеллмана (Diffie-Hellman)

Симметричные алгоритмы шифрования (например, DES, 3DES, AES и др.) требуют для своей работы один и тот же ключ для шифрования и дешифрования данных. Диффи и Хеллманом была поставлена задача выработки совместного секретного ключа, пользуясь только открытыми (т.е. незащищёнными) каналами связи. Это значит, что атакующий, даже перехватив весь обмен данными между двумя системами, не сможет за разумное (полиномиальное) время вычислить общий секретный ключ. Алгоритм Диффи-Хеллмана выглядит так:

1. Зная основание – большое простое число *p*, и некоторый порождающий элемент группы

$\alpha \in G_p^*$ стороны выбирают каждая свой неразглашаемый ключ (это служебное число, оно не используется для шифрования/дешифрования трафика) – числа x_A, x_B соответственно
2. Далее, стороны вычисляют следующие (открытые) значения
$y_A = \alpha^{x_A} \bmod p$
$y_B = \alpha^{x_B} \bmod p$
и посылают друг другу эти значения.
3. Далее, на основании имеющегося у каждой из сторон своего закрытого и чужого открытого значений (например, А знает x_A, y_B) каждая сторона вычисляет закрытый общий ключ *K* по формуле:
$K = y_B^{x_A} \bmod p = (\alpha^{x_B} \bmod p)^{x_A} = \alpha^{x_B x_A} \bmod p = y_A^{x_B} \bmod p$
Легко видеть, что обе стороны вычислят один и тот же ключ.

Проблема в программной реализации этой идеи состоит в том, чтобы обе стороны использовали одни и те же значения p, α . Решением, представляемым, например, компанией Cisco, является предварительный обмен числами *p* и *q* . Число *p* используется как модуль, а на основании *q* вычисляется общий элемент α.
Как вы уже видели, алгоритм возведения числа в степень по модулю *p* – это быстрый алгоритм (количество операций - $\underline{\underline{O}}(\ln p)$), а вот эффективного алгоритма для дискретного логарифмирования пока не найдено, поэтому пересылка открытых значений *y* безопасна.

Глава 3. DSS

Digital Signature Standard (американский Стандарт Цифровой Подписи) – способ, с помощью которого можно не только гарантировать целостность сообщения, передаваемого через открытые сети, но и удостоверяться, что сообщение послано легитимным (полномочным) отправителем. Идея алгоритма, использующего идею распределённых ключей (т.е. один открытый и общедоступный, а другой – закрытый и личный) проста: если зашифровать сообщение своим закрытым ключом, то любой получатель сможет, обратившись к хранилищу открытых ключей, проверить подлинность отправителя, расшифровав сообщение открытым ключом отправителя.

Стандарт DSS в качестве алгоритма шифрования использует DSA (Digital Signature Algorithm). DSA генерирует ключи длиной от 512 до 1024 бит следующим образом:

1. Выбирается простое число *q*, такое что $2^{159} < q < 2^{160}$
2. Выбирается *t* такое что $0 \le t \le 8$
3. Выбирается простое *p* так, что $2^{511+64t} < p < 2^{512+64t}$, причём *q* делит *(p-1)*, т.е. *(q, p-1)=q*
4. В циклической мультипликативной группе Z_p^* находится элемент α порядка *q*, для этого выбирается $g \in Z_p^*$ $\alpha = g^{(p-1)/q} \bmod p$. Если $\alpha \neq 1$, то он является искомым элементом. Заметим, что этот элемент является порождающим для циклической подгруппы <α> порядка *q*.
5. Выбирается случайное целое *b* такое что: $1 < b \le q - 1$
6. Вычисляется $y = \alpha^b \bmod p$
7. Открытый ключ – *A=(p,q,α,y)*, закрытый – *b*.

Чтобы подписать некоторое сообщение требуется выполнить следующие шаги:

1. Выбрать произвольное k, $0<k<q$
2. Вычислить $r = (\alpha^k \bmod p) \bmod q$
3. Вычислить $k^{-1} \bmod q$
4. Вычислить $s = k^{-1}(h(m) + \alpha * r) \bmod q$, где *h(m)* – значение хэш-функции от сообщения *m*
5. Подписью *A* для *m* является пара *(r,s)*

Глава 4. Алгоритмы шифрования DES, 3DES, AES

DES – Data Encryption Standard – старый американский стандарт шифрования. Является так называемым блочным потоковым шифром, т.е. работающим с блоком данных некоторой длины. Для DES этот блок составляет 64 бита, длина ключа для шифрования – 56 бит плюс 8 бит контрольной суммы.
Алгоритм DES получает на входе блок данных длиной 64 бита, ключ для шифрования и проводит последовательно 16 раундов обработки блока. Каждый раунд содержит битовые операции ключа и данных, а также перестановку половин 64-битового блока. Расшифровка производится аналогично с применением того же ключа.

На данный момент длины ключа в 56 бит не достаточно, ибо даже метод грубой силы (полного перебора) даёт результат за приемлемое время (за день). Был разработан модифицированный алгоритм.

3DES – Triple DES – алгоритм, использующий ключ длиной 168 бит, состоящий из трёх блоков по 56 бит. Этот алгоритм тоже поточный и тоже блоковый. Блок составляет, как и у DES 64 бита. Для зашифровывания 64 бит текста три раза применяется алгоритм DES каждый раз с новой третью ключа. Конечно, такой шифр сломать труднее, но есть предположение, что криптостойкость его не сильно выше, чем у DES. При этом очевидно, что производительность шифрующего оборудования сильно падает, по сравнению с DES (примерно в три раза)

В 2000 году завершился американский 2-хлетний конкурс на новый стандарт шифрования. Его победителем стал алгоритм Rijndael. Он и был назван новым стандартом.

AES – Advanced Encryption Standard – новый американский стандарт шифрования. Блочный потоковый шифр. Использует 128-битный ключ (есть варианты с 192 и 256 битовыми ключами) и работает с блоком длиной 128 бит (192 и 256 бит соответсвенно). В основу положена идея побитового сложения с ключом блока текста и перестановки строк и столбцов текста, представленного в виде матрицы (один раунд). Выполняется 10 раундов, при этом достаточная криптостойкость достигается уже при 8 раундах, а дополнительные 2 раунда лишь усиливают её. Данный алгоритм быстр, достаточно прост в реализации и требует мало памяти и ресурсов процессора, что особенно важно при использовании его в смарт-картах и других системах с ограниченными ресурсами (restricted space).

Глава 5. Центры сертификатов (Certificate Authority)

Для дальнейшего рассмотрения давайте разберемся, как работает схема цифровой подписи. Из Главы II вы уже знаете, что сообщение, зашифрованное открытым ключом, может быть расшифровано закрытым. Но верно и обратное: то, что зашифровано закрытым ключом можно расшифровать открытым (элементарное доказательство , оставляю на Вас, читатель). Но зачем использовать такую схему? Она используется для подтверждения того факта, что именно указанный отправитель прислал вам сообщение и оно не было видоизменено в процессе передачи. Делается это следующим образом:

Рассмотрим самую распространенную схему – у каждой из сторон по паре ключей. В каждой паре один ключ называется открытым и его можно передавать по незащищенным каналам связи, а второй – закрытым и его никогда не передают по открытым канал связи.

Пусть у нас есть ключи хоста A: открытый E(A) (encrypt) и D(A) (decrypt)
Обозначим преобразование входного массива данных на ключе K как
$[\text{массив}]_K$

Примечание: чтобы излишне не усложнять дальнейшее изложение, не будем разделять в условных обозначениях шифрование и расшифровывание. К тому же часто на самом деле это одна и та же процедура, только выполняющаяся на разных ключах.

Тогда для произвольной строки text выполняется равенство:
$\mathbf{[[text]_{E(A)}]_{D(A)}=text}$

И наоборот
$\mathbf{[[text]_{D(A)}]_{E(A)}=text}$

Эти 2 равенства дают нам возможность как производить шифрование данных, предназначенных конкретному получателю, так и удостоверять свою «личность», используя «секрет», известный только мне (мой закрытый ключ), добавляя к передаваемому сообщению проверочные данные, зашифрованные моим закрытым ключом (электронная цифровая подпись, ЭЦП).

Пример:
Пусть хост A хочет передать хосту B некие конфиденциальные данные (text). Тогда он должен каким-либо образом раздобыть открытый ключ хоста B (как это сделать в реальных сетевых условиях рассмотрим позже, а пока предположим, что есть специальная «доска объявлений», на которой «приколоты» все нужные открытые ключи). Дальше нужный текст шифруется
$\mathbf{cipher_B=[text]_{E(B)}}$
и полученный шифр передается по сети. Не имея закрытого ключа хоста B «взломать» шифр практически невозможно (вернее, невозможно за разумное время с большой вероятностью. Подробнее о том, почему RSA сложный для взлома можно прочитать в литературе по RSA. Можно также ознакомиться в брошюре …)
На принимающей стороне хост B производит расшифровывание и получает исходный текст:
$\mathbf{[cipher_B]_{D(B)} = text}$

В приведенном примере есть одна тонкость: хост B пока никак не может удостовериться, что именно хост A послал ему сообщение, потому что никакой возможности явной проверки нет.

Остается только верить :) Чтобы предоставить хосту B возможность проверки источника, необходимо не просто передать текст, а добавить к нему некую проверочную последовательность. Можно делать так:
$\mathbf{text+[text]_{D(A)}}$

Однако, понятно, что при большом объеме данных (text) шифр будет такого же размера и считать его долго. С другой стороны, нам нет необходимости полностью шифровать текст с единственной целью удостоверения отправителя. Нам достаточно зашифровать лишь некую проверочную последовательность. В качестве такой последовательности используется стандартная функция хэширования (sha или md5). Т.е. если стоит задача только удостоверить передающую сторону, делается следующее (ЭЦП):
$\mathbf{text+[hash[text]]_{D(A)} = text+ctrl_A}$

На принимающей стороне отделяются последние 128 (md5) или 160 (sha) бит, остаток хэшируется и производится сравнение:
$\mathbf{hash[text]\ V\ [ctrl_A]_{E(A)}}$

Если в процессе передачи text не изменялся, а также источник именно тот, за кого себя выдает, то легко видеть, что будет равенство (мы по-прежнему подразумеваем наличие общей «доски объявлений» с открытыми ключами)
$\mathbf{hash[text]=[ctrl_A]_{E(A)}}$

Ну а если требуется одновременно решать обе эти задачи, т.е. и передавать безопасно конкретному получателю и подтверждать источник, то обе технологии совмещают, т.е. берется text и сторона A делает сообщение:
$\mathbf{[text]_{E(B)}+[hash[[text]_{E(B)}]]_{D(A)}= cipher_B+[hash[cipher_B]]_{D(A)}= cipher_B+ctrl_A}$

На принимающей стороне отделяется $ctrl_A$, расшифровывается открытым ключом A. Остаток хешируется и полученные части сравниваются.

Цифровые сертификаты X.509v3

Пока мы оставляли в стороне вопрос «доски объявлений», а именно, насколько можно доверять тому, что на ней опубликовано? Злоумышленник вполне мог вместо легитимного открытого ключа подставить свой и спокойно обманывать всех.

Второй вопрос: а как с сетевой точки зрения сделать такую «доску объявлений?

Обе эти задачи одним махом решаются системой удостоверяющих центров (эдаких сетевых «нотариусов»), которые гарантируют принадлежность открытого ключа.

Каждый удостоверяющий центр (Certification Authority, CA) имеет свою пару ключей E(CA) и D(CA). Эта пара неизменна в течение всей жизни данного CA. CA вырабатывает сам себе сертификат. Формат сертификата зависит от стандарта. Рассмотрим самый распространенный формат: X.509v3
Сертификат - это в-основном хранилище и «переносчик» открытого ключа. «Бумажка», удостоверяющая владельца. В сертификате обязательно содержатся следующие поля:

1. Время жизни сертификата. Это поле проставляет сертификатный сервер в соответствии со своими настройками.
2. Кому выдан (поле subject). В нем содержатся данные владельца сертификата, например, имя, департамент, организация, e-mail и т.д. Эти параметры передаются тем, кто запрашивает сертификат
3. Кем выдан (поле issuer). Данные выдавшего сертификатного сервера. Имя, а также дополнительные параметры. Заполняется сервером сертификатов
4. Открытый ключ владельца. Передается владельцем при запросе на изготовление сертификата.
5. Цифровая подпись. Весь выданный сертификат со всеми полями подписывается электронной цифровой подписью сервера сертификатов, тем самым удостоверяя все данные. Для надежности может даже быть 2 цифровых подписи: с хэшем md5 и sha.

Сервер сертификатов вырабатывает себе так называемый самоподписанный (корневой, root) сертификат. У этого сертификата есть отличительная особенность: поле subject равно полю issuer. Запишем тело сертификата как открытый ключ плюс атрибуты. Тогда схематически это можно описать так :

$$\mathbf{cert_{CA}=E(CA)+ATTR_{CA}+[hash[E(CA)+ATTR_{CA}]]_{D(CA)}}$$

Как видите, в корневом сертификате используется свой же закрытый ключ. А это значит, что такой сертификат может себе изготовить каждый. Именно поэтому все браузеры ругаются (предупреждают об опасности) при получении от сайта по https самоподписанный сертификат в качестве удостоверяющего, ибо его никак не получится проверить. Останется только переложить ответственность на пользователя, чтобы тот дрожащей рукой ткнул в «все равно продолжить» на свой страх и риск :)

Поэтому-то нам важен этап аутентификации самого сервера, чтобы никакой злоумышленник не подменил идентификатор сервера в своих гнусных целях. Аутентификация сервера может производиться разными способами. Например, может каким-либо несетевым способом доставляться корневой сертификат сервера CA на клиентскую машину. Курьером/голубями/на бумажке. Либо проверочная сумма сертификата (как правило, хэш корневого сертификата – fingerprint) лежит на закрытой странице сайта или её можно проверить по телефону.

После того, как мы стали доверять центру сертификатов, т.е. каким-либо образом получили его корневой сертификат и положили в хранилище доверенных корневых сертификатов, наступает этап запроса на выработку нам сертификата данным сервером сертификатов (enrollment).

На этом этапе запрашивающий хост создает стандартный запрос, в который включает свой открытый ключ, атрибуты поля subject (имя, департамент, e-mail и т.д.) отправляет свой запрос. Либо по сети (по протоколу SCEP – Simple Certificate Enrollment Protocol), либо используя GUI сервера, заполняя соответствующие поля запроса или напрямую передавая ему запрос в виде файла. В любом случае все атрибуты попадают на сервер сертификатов, и тот вырабатывает нам наш собственный (идентифицирующий, identity) сертификат:

$$\mathbf{cert_{A}=E(A)+ATTR_{A}+[hash[E(A)+ ATTR_{A}]]_{D(CA)}}$$

Как видно, наш сертификат уже не самоподписанный, а защищен ЭЦП сервера сертификатов.

Таким образом, на каждом хосте для их взаимодействия, должно быть 2 сертификата: корневой сертификат используемого сервера сертификатов, которому мы доверяем, а также наш собственный, идентифицирующий сертификат, подписанный сервером. Тогда если 2 хоста, A и B хотят пообщаться, происходит обмен сертификатами и следующие 3 проверки:

1. Идентифицирующий сертификат должен быть действителен. Время жизни не должно истечь. А проверяем мы по своим локальным часам, значит часам надо уделить пристальнейшее внимание: сбой сервера ntp или просто нарушение даты может перечеркнуть все наши старания при работе с сертификатами.
2. Сертификат должен быть подписан сервером, которому мы доверяем. И неизменен. Пусть сертификат хоста B выглядит так:

$\mathbf{cert_B = E(B)+ATTR_B+[hash[E(B)+ ATTR_B]]_{D(CA)}= E(B)+ATTR_B+ctrl_B}$

Тогда на хосте A проводим проверку: отделяем поле цифровой подписи ($ctrl_B$), расшифровываем его, используя корневой сертификат, хэшируем остаток и сравниваем

$\mathbf{hash[E(B)+ ATTR_B]\ V\ [ctrl_B]_{E(CA)}}$

Если полученный сертификат подписан кем надо и не изменялся после выдачи, то будет равенство:

$\mathbf{hash[E(B)+ ATTR_B] = [ctrl_B]_{E(CA)}}$

3. И ещё можно проводить проверку, не отозван ли присланный сертификат по каким-то причинам. Для этого на сервере сертификатов хранится список отозванных сертификатов (Certificate Revocation List, CRL). Это просто перечисление номеров всех отозванных сертификатов. Посмотреть может каждый. Но этот список подписан ЭЦП сервера сертификатов, поэтому его всегда можно проверить, не изменен ли он злонамеренно.

Здесь уместно отметить, что поиск корневого сертификата для проверки ЭЦП ведется по имени (subject). Поэтому не удивительно, что указав, например, в браузере адрес, а не имя, вы получите предупреждение, что от данного хоста получен сертификат, выданный другому и ему нет доверия. Конечно, сертификат выдан для хоста с именем (CN), а обращаемся мы на ip адрес.

И поле сертификата со ссылкой на список отозванных сертификатов содержит имя сервера. Поэтому часто в корпоративных серверах CA бывает проблема – не доступен CRL. А все потому, что при установке серверу задали имя, которого нет в DNS. Это поле можно посмотреть и прописать в DNS.

Вот собственно и вся инфраструктура. Можно добавить, что кроме основного корневого центра сертификатов могут существовать еще целая цепочка подчинённых (Registration Authority, RA), которых может быть много для распределения нагрузки. Сертификат RA неразрывно связан с сертификатом CA и часто вся цепочка сертификатов пересылаются клиенту. И список отозванных сертификатов тоже для распределения нагрузки, можно положить не только на сам сервер, но и на другие сервера, доступные по http, ldap или scep. Такие сервера будут называться серверами раздачи списка отозванных сертификатов (CRL Distribution Points).

Если же вы хотите организовать сервис, которому все будут доверять (например, сайт, опубликованный по протоколу https), то либо надо всем клиентам раздать корневой сертификат вашего собственного сервера сертификатов, либо использовать заложенные

по умолчанию в большинство ОС корневые сертификаты каких-нибудь грандов (в системах Windows их можно посмотреть в закладке «Доверенные центры сертификации» в оснастке работы с сертификатами). Правда, изготовленный для вас сертификат уже не будет бесплатным. :)

ЧАСТЬ III. IPSec

Глава 1. Что такое IPSec

Бурное развитие сетевых технологий и проникновение сетей в повседневную жизнь привело к тому, что всё больше и больше пользователей стараются использовать незащищённые сети передачи данных для своих приватных нужд, как то: связывание частных сетей через общие, передача голоса и т.д. Идея использовать публичные сети основывается на том, что для реализации различных технологий через них требуется гораздо меньше средств, чем при классических решениях. Также на использование открытых сетей влияет бурное развитие электронных бизнес-приложений, как то кредитные карточки, удалённое управление банковским счётом, электронные системы банк-клиент и т.д. Поэтому возникла острая необходимость каким-либо образом стандартизировать методы защиты в публичных сетях передачи данных.

Одним из стандартов является технология шифрования данных на третьем уровне модели OSI с использованием различных протоколов аутентификации, шифрования, обмена ключами и т.д. Этот стандарт (а вернее сказать, набор стандартов) назвали IPSec. В него входят:

1. Стандартные протоколы хэширования: SHA, MD5
2. Стандартные алгоритмы шифрования: DES, 3DES, AES
3. Протоколы передачи шифрованных данных над IP: ESP, AHP
4. Протокол генерирования общего закрытого ключа: Diffie-Hellman
5. Протокол обмена ключами IKE

Комбинирование этих протоколов задаёт весь процесс создания шифрованного туннеля между двумя точками. Процесс создания туннеля можно разделить на несколько шагов (в литературе чаще встречается понятие «фаз IKE» - «IKE Phase I» и «IKE Phase II»)

1. Создание первичного (IKE Phase I) шифрованного туннеля с аутентификацией для последующего использования его для согласования параметров основного туннеля. На этом этапе согласовываются следующие параметры:
- способ аутентификации: предустановленным ключом или с помощью сертификатов
- способ шифрования: DES, 3DES
- способ хэширования: SHA или MD5
- способ генерирования общего ключа алгоритмом Diffie-Hellman: ключ длиной 768 (группа 1), 1024 (группа 2) или 2048 (группа 5)
- время жизни этого туннеля

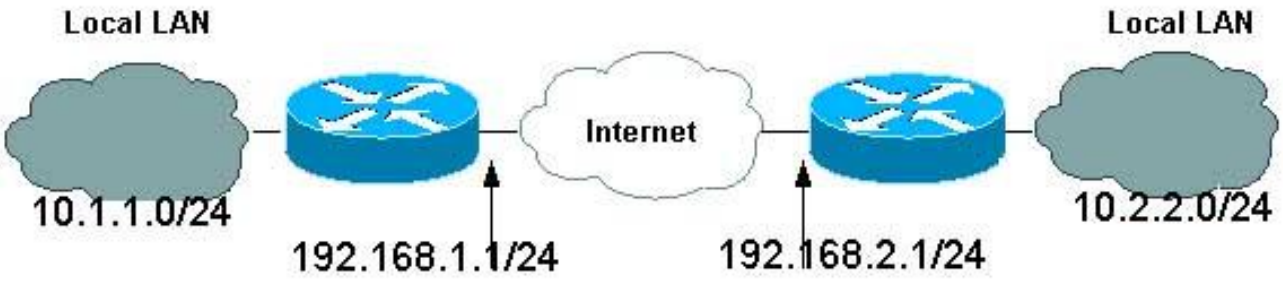
2. Создание основного туннеля для защищённой передачи данных (IKE Phase II). В него входят следующие параметры:
- способ преобразования трафика, т.е. каким по какому протоколу будут идти зашифрованные данные, по ESP (Encapsulated Security Payload), который шифрует весь пакет и добавляет новый ip-заголовок, или по AHP (Authenticated Header Protocol), который лишь добавляет проверочный хэш в заголовок пакета, не шифруя содержание. А также, какой алгоритм шифрования будет использоваться (DES, 3DES,

AES, NULL). Здесь же можно скомбинировать методы: не только шифровать, но и защищать зашифрованное хэшем MD5 или SHA.
- Время жизни основного туннеля.
- способ генерирования общего ключа алгоритмом Diffie-Hellman: ключ длиной 768 (группа 1), 1024 (группа 2) или 2048 (группа 5)

Используя шифрованные туннели IPSec пользователи могут не опасаться за конфиденциальность данных, пересылаемых через публичную сеть, поэтому они могут логически объединить свои частные (локальные) сети, создавая единую логическую частную сеть. Такая технология называется VPN – Virtual Private Network, виртуальная частная сеть. Второй возможностью является подключение удалённых клиентов, используя специальное программное обеспечение (VPN-клиент) на клиентской стороне и специальные настройки на оборудовании доступа. Таким образом удалённые клиенты смогут попадать в частную сеть организации, работать со своими приватными ресурсами там, не опасаясь за их огласку.

Глава 2. Настройка Site-to-Site VPN на маршрутизаторах Cisco

В данной главе я опишу процесс настройки маршрутизатора Cisco (например, 26xx, 17xx с версией операционной системы Cisco IOS не ниже 12.2.8T) для создания шифрованного туннеля, соединяющего две частные сети через публичную сеть Internet.
Замечание: после значка «!» идут комментарии к последующей строке.



RouterA# conf t
! Создание политики безопасности под номером 10 (IKE Phase I)
RouterA(config)# crypto isakmp policy 10
! Выбирается алгоритм хэширования
RouterA(config-isakmp)# hash sha
! Выбирается алгоритм шифрования
RouterA(config-isakmp)# encryption aes
! Выбирается способ первичной аутентификации соседей. Альтернативой является rsa-sig,
! задающей использование сертификатов. Этот способ идёт по умолчанию.
RouterA(config-isakmp)# authentication pre-share
! Задаёт длину ключа Diffie-Hellman
RouterA(config-isakmp)# group 2
! Задаёт время жизни первичного туннеля
RouterA(config-isakmp)# lifetime 86400
RouterA(config-isakmp)# exit
! Задаёт пароль для аутентификации с удалённым соседом с адресом 192.168.2.2

RouterA(config)# crypto isakmp key MYKEY address 192.168.2.2
!
! Следующие шаги задают создание основного туннеля (IKE Phase II)
!
! Задаёт алгоритм шифрования, протокол передачи и хэширование (не обязательный
! параметр). Здесь задаём ip-протокол ESP с шифрованием DES и дополнительным
! хэшированием SHA.
RouterA(config)# crypto ipsec transform-set MYSET esp-aes esp-sha-hmac
! Создаём список доступа для описания трафика, который пойдёт в туннель
RouterA(config)# access-list 101 permit ip 10.1.1.0 0.0.0.255 10.2.2.0 0.0.0.255
! Создаём крипто-карту (список параметров), которую впоследствии применим на
! интерфейсе. Номер 10 – 10-й абзац крипто-карты. Абзацы упорядочены по номеру.
RouterA(config)# crypto map MYMAP 10 ipsec-isakmp
! Указываем тот трафик, который пойдёт в туннель (и инициирует его создание)
RouterA(config-crypto-map)# match address 101
! Задаём применение правила преобразования и передачи трафика через туннель
RouterA(config-crypto-map)# set transform-set MYSET
! Задаём время жизни основного туннеля
RouterA(config-crypto-map)# set security-association lifetime seconds 86400
! Указываем соседа, с кем устанавливается шифрованный туннель
RouterA(config-crypto-map)# set peer 192.168.2.2
RouterA(config-crypto-map)# exit
RouterA(config)# interface serial 0/0
! Задаём применение крипто-карты на интерфейсе
RouterA(config-if)# crypto map MYMAP

Приведу также простейшую настройку Cisco ASA для работы с site-to-site IPSec VPN. Часть настроек буду очень похожи на настройки маршрутизатора, часть — нет.

AsaB# conf t
! Создание политики безопасности под номером 10 (IKE Phase I)
AsaB(config)# crypto isakmp policy 10
! Выбирается алгоритм хэширования
AsaB(config-isakmp)# hash sha
! Выбирается алгоритм шифрования
AsaB(config-isakmp)# encryption aes
! Выбирается способ первичной аутентификации соседей. Альтернативой является rsa-sig,
! задающей использование сертификатов. Этот способ идёт по умолчанию.
AsaB(config-isakmp)# authentication pre-share
! Задаёт длину ключа Diffie-Hellman
AsaB(config-isakmp)# group 2
! Задаёт время жизни первичного туннеля
AsaB(config-isakmp)# lifetime 86400
AsaB(config-isakmp)# exit
! Включается протокол ISAKMP на интерфейсе outside. По умолчанию он выключен
AsaB(config)# isakmp enable outside
! Создаётся туннельная группа для соседа
AsaB(config)# tunnel-group 192.168.1.2 type ipsec-l2l
! В режиме ввода IPSec параметров задаются настроки для соседа

AsaB(config)# tunnel-group 192.168.1.2 ipsec-parameters
! Задаёт пароль для аутентификации с удалённым соседом с адресом 192.168.1.2
AsaB(config-tunnel)# pre-shared-key *<ключ>*
!
! Следующие шаги задают создание основного туннеля (IKE Phase II)
!
! Задаёт алгоритм шифрования, протокол передачи и хэширование (не обязательный
! параметр). Здесь задаём ip-протокол ESP с шифрованием DES и дополнительным
! хэшированием SHA.
AsaB(config)# crypto ipsec transform-set MYSET esp-aes esp-sha-hmac
! Создаём список доступа для описания трафика, который пойдёт в туннель
AsaB(config)# access-list 101 permit ip 10.2.2.0 255.255.255.0 10.1.1.0 255.255.255.0
! Создаём крипто-карту (список параметров), которую впоследствии применим на
! интерфейсе. Номер 10 – 10-й абзац крипто-карты. Абзацы упорядочены по номеру.
AsaB(config)# crypto map MYMAP 10 ipsec-isakmp
! Указываем тот трафик, который пойдёт в туннель (и инициирует его создание)
AsaB(config)# crypto map MYMAP 10 match address 101
! Задаём применение правила преобразования и передачи трафика через туннель
AsaB(config)# crypto map MYMAP 10 set transform-set MYSET
! Задаём время жизни основного туннеля
AsaB(config)# crypto map MYMAP 10 set security-association lifetime seconds 86400
! Указываем соседа, с кем устанавливается шифрованный туннель
AsaB(config)# crypto map MYMAP 10 set peer 192.168.2.2
! Задаём применение крипто-карты на интерфейсе
AsaB(config)# crypto map MYMAP interface outside
! Если используется NAT, то надо пробросить пакеты, предназначенные для туннеля, мимо
NAT. Для этого используется NAT 0
! Создание спсика доступа для NAT 0
AsaB(config)# access-list NONAT permit ip 10.2.2.0 255.255.255.0 10.1.1.0 255.255.255.0
! И применение его
AsaB(config)# nat (inside) 0 access-list NONAT

Приложение 1. Ответы на задачи.

Задача 1: жезл определённой толщины, очевидно, задаёт шаг в тексте, с которым надо брать буквы, чтобы они сложились в слова. Понятно, что лента была намотана хотя бы 2 раза (не случайно есть оговорка про однобуквенные слова), поэтому величина шага не может превышать *N/2*, т.е. атакующему будет достаточно *N/2* шагов для взлома шифра. Конечно, это далеко не самая лучшая оценка количества попыток, однако и она достаточна хороша.

Задача 2: $\varphi(30) = 8, \varphi(1024) = 512, \varphi(200560490130) = 30656102400$

Пояснение: конечно, для первого значения можно посчитать в лоб. Однако, уже для второго значения придётся применять формулу. Правда, здесь можно догадаться: взаимно простыми с 1024 будут все нечётные, коих ровно половина. Ну а для нахождения последнего значения придётся разложить на множители. Число легко раскладывается на множители и по известной формуле легко посчитать требуемое значение. Простота разложения большого числа на простые сомножители здесь основана на том, что все делители невелики. Есть быстрые алгоритмы, раскладывающие на множители большие числа, если все простые составляющие относительно невелики. Отсюда ясно, почему разработчики RSA предлагают использовать большие простые сомножители – чтобы не упрощать задачу разложения на множители.

Задача 3: Для начала надо посчитать, какое максимальное значение может принимать число длиной 512 (768, 1024, 2048) бита. Для этого вспомним двоичное представление. Легко видеть, что минимальное число равно 2^{512}. Для его разложения необходимо перебирать все простые числа вплоть до $\sqrt{2^{512}} = 2^{256}$. По приведенной формуле, простых чисел на этом промежутке не меньше, чем $\frac{2^{256}}{\ln 2^{256}} = \frac{2^{248}}{\ln 2} \le 6.55 * 10^{74}$

Соответственно для 768 бит: $1{,}48 * 10^{113}$

Для 1024 бит: $1{,}14 * 10^{151}$ и т.д.

Приложение 2. Применяемые обозначения

Далее для справки приводятся некоторые общепринятые в математике обозначения, применяемые и в этом пособии

Обозначение	Расшифровка
$\forall$	«Для любого»
$\exists$	«Существует»
$\in$	«Принадлежит»
!	«Единственный»
$\Rightarrow$	«Следовательно»
lim	«Предел»
| A |	«Мощность множества А», т.е. количество элементов в нем
ln	«Натуральный логарифм», т.е логарифм по основанию ***e***
$[Text]_{E_A}$	Шифр сообщения *Text* открытым ключом хоста *A*
hash[Text]	Хэш сообщения *Text*
$[Text]_1+[Text]_2$	Присланное сообщение состоит из двух частей, который можно однозначно отделить друг от друга.

Пример:

$$Если\ \forall\ \varepsilon > 0, b \in A\ \exists! a \in A : a^{\varepsilon} = b \Rightarrow A - замкнуто$$

Читается так:
Если для любого эпсилон, большего нуля и любого *b*, принадлежащего множеству *A*, существует единственное *a* принадлежащее множеству *A*, следовательно *A* – замкнуто.

СПИСОК ЛИТЕРАТУРЫ

1. «Курс алгебры.» Э.В.Винберг. 3-е издание, испр. и доп.. Издательство «Факториал Пресс», 2002
2. «Введение в теорию чисел. Алгоритм RSA.» С. Коутинхо. Издательство «Постмаркет», 2001г.
3. «Введение в криптографию». Под общей редакцией В.В.Ященко. 3-е издание, дополненное. Издательство «ЧеРо», 2000г.

Интернет-ресурсы

1. http://se.math.spbu.ru/Courses/Crypto/Crypto.pdf
2. http://www.morepc.ru/security/crypt/lan200105110.html