Криптография и Golang

I. КРИПТОГРАФИЯ

С момента рождения письменности, у людей появлялась острая необходимость скрывать сообщения любыми доступными способами от лиц, которым оно не предназначалось. И в это самое время появляется такая наука, как криптография, решающая эти проблемы.

1. Введение

Криптография является одной из самых древних наук, решающая, в настоящее время, не только проблемы сокрытия информации, но также и ряд других проблем: тайного голосования, обмена ключей, проверки неизменности сообщения, невозможности отказа от авторства и т.д.

Сама наука криптография, очень часто пересекается с такими науками как история, информатика и непосредственно математика.

Часто криптографию разделяют на две части. Первая часть криптографии – классическая, иначе говоря вся та криптография, которая работала не с байтами и битами, а с буквами, словами и предложениями. Вторая часть криптографии – современная, которая использует уже компьютерные технологии для работы с числами на уровне битов и байтов.

2. Терминология

Терминологий в криптографии достаточно много, если углубляться в детали, тем не менее, на этапе начала, следует рассказать только об основных.

1. *Открытое сообщение* – информация, которая имеет осмысленные данные. Часто в протоколах такую информацию подвергают шифрованию.

2. *Шифрование* – способ скрытия информации, под средством перевода её в нечитаемый вид, где используется определённый ключ шифрования. На выходе получаем закрытое сообщение.

3. *Расшифрование* – противоположные действия шифрованию, иными словами перевод из закрытого сообщения в открытое под средством использования ключа.

4. *Ключ шифрования* – элемент определённого шифра, при помощи которого возможно шифровать и/или расшифровывать информацию. По уровню важности такой же как и само открытое сообщение.

5. *Шифр* – алгоритм или же функция (с математической точки зрения), которая принимает на входные параметры открытое/закрытое сообщение и ключ. На выходе получаем закрытое/открытое сообщение.

Закрытое сообщение часто также назвают криптограммой или зашифрованным сообщением. Шифр также могут называть криптографическим алгоритмом или алгоритмом шифрования.

В русской криптографии сложилась некая особенность на счёт двух слов: расшифрование и дешифрование. Как говорилось ранее, расшифрование – это перевод из закрытого сообщения в открытое, при помощи ключа. Дешифрование же является расшифрованием без использования ключа, иными словами – взлом шифра.

6. *Криптоанализ* – отдельная часть криптографии, в которой изучаются методы взлома шифров, протоколов, нахождения их уязвимостей и временные рамки надёжности.

7. *Стеганография* – способ хранения информации, под средством сохранения в тайне самого факта о присутствии данной информации.

8. *Симметричные алгоритмы* – криптографические алгоритмы, в которых ключ служит как для шифрования, так и для расшифрования информации.

9. *Асимметричные алгоритмы* – криптографические алгоритмы, в которых присутствует два ключа – открытый (для шифрования) и закрытый (для расшифрования).

10. *Криптостойкость* – характеристика определённого алгоритма шифрования, в котором выявляются его слабости, как со стороны теории, так и со стороны практики. И на основе этого, дают ему определённый уровень защищённости.

11. *Кодирование* – шифрование отдельных слов, фраз по справочнику (ключу) на другие слова и фразы.

3. Математическое описание

При помощи математики достаточно легко и наглядно описать работу различных протоколов или же отдельные части криптографического алгоритма, тем не менее, не стоит расписывать таким же способом полностью шифры. Гораздо нагляднее описать их при помощи пошагового алгоритма или же вовсе расписать всю работу от А до Я.

Часто криптографы предоставляют следующие данные и функции:

M – открытое сообщение
C – закрытое сообщение
K – ключ шифрования
E – функция шифрования
D – функция расшифрования
H – криптографическая хеш-функция
S – цифровая подпись

Часто как в симметричных, так и в асимметричных системах шифрования, функции E и D коммутативны, иными словами
D(E(M)) = E(D(M)) = M

Также существуют такие алгоритмы, в которых функция шифрования равна функции расшифрования E = D, таким образом выстраивается следующая формула:
E(E(M)) = D(D(M)) = M

Иногда применяют к алгоритмам, у которых нет свойства повторяемости, тройное шифрование, где используется два ключа:
$E_{K1}(D_{K2}(E_{K1}(M))) = C$
Такая схема использовалась в алгоритме 3DES, которая увеличивала ключ с 56 бит до 112.

Криптографы помимо самих данных и функций, указывают также и участников протокола:
A – Алиса (Alice) – отправляющий сообщение
B – Боб (Bob) – принимающий сообщение
E – Ева (Eve) – перехватчик сообщений (только чтение данных)
M – Мэлори (Malory) – человек посередине (MITM атака)
T - Трент (Trent) – посредник между Алисой и Бобом, доверенное лицо.

Имея вышеперечисленные данные, можно легко составить на основе этого простой протокол:

1. [A] K → K → K [B]
2. [A] M → $E_K(M)$ → C → $D_K(C)$ → M [B]

где Алиса и Боб имеют общий ключ K и договорённость на счёт алгоритмов E и D.
Этот протокол имеет уязвимости, в которых как Ева, так и Мэлори способны совершать определённые атаки.

Ева способна перехватить ключ шифрования и впоследствии читать всю переписку.
Мэлори способен перехватить ключ шифрования, а потом прикидываться Алисой, либо же Бобом для выявления нужной ему информации. Либо же, на этапе обмена ключей, подкинуть свой ключ.
Также в данном протоколе, играет не последнюю роль сам криптографический алгоритм. Если шифр не криптостойкий, то существует большой успех криптоанализа.
Помимо всего прочего, сам ключ может быть также слабым звеном протокола.
Таким образом, стойкость протокола ориентируется на его самое слабое звено. Не важно на сколько будет продвинутой и идеальной сама схема протокола, если ключ был выбран ненадёжным.

4. Криптостойкость

Криптографическая стойкость алгоритмов делится на три уровня:
1. Некриптостойкие
2. Вычислительно криптостойкие
3. Абсолютно криптостойкие

К некриптостойким относятся такие методы шифрования, которые имеют определённые уязвимости и непригодны для шифрования сообщений в настоящих или ближайших будущих реалиях. Так например, уже непригодны для использования такие шифры, которые в своё время были вычислительно криптостойкими, например шифр DES.

К вычислительно криптостойким относят шифры, которые ориентированы для использования не только в настоящем времени, но и с заделом на будущее. К таким шифрам можно отнести AES, RSA.
И как ни странно, уже давно доказаны и предложены алгоритмы с абсолютной криптостойкостью, где даже имея все ресурсы мира, невозможно вычислить исходное сообщение не имея при этом ключа. Такие шифры называют одноразовыми блокнотами, к которым относится шифр Вернама.

Принцип работы одноразовых блокнотов, на примере шифра Вернама, следующий:

1. Существует открытое сообщение M длиной в L байтов.

2. В качестве ключа K генерируется истинно случайная последовательность байтов длинной в L, то-есть L(K) = L(M).

3. Производится гаммирование M и K образуя C.

4. После завершения протокола, ключ K уничтожается и заного не используется.

Вся суть в том, что ключ является истинно случайным, где любая вероятность подобранного ключа будет равняться вероятности любого иного события.

Тем самым, если сообщение M находится в диапозоне от A до Z и равняется трём байтам, то возможных результатов будет $26^3 = 17576$ и любой вариант из этого списка будет равновероятным. То-есть, может быть сообщение M равняться CAT, может быть и DOG, а может и вовсе RED и тд.

Но у этого алгоритма есть определённые негативные свойства, которые перечёркивают даже абсолютную криптостойкость:

1. Длина ключа равняется длине сообщения. Если сообщение равно 10Гб, то и ключ обязан быть равен 10Гб.

2. Сложность генерирования истинно случайной последовательности. Если ключ сгенерирован при помощи псевдослучайного генератора, даже криптографического, то уже существует уязвимость, где криптоаналитик будет направлять все силы на взлом этого самого генератора.

3. Невозможность использования ключа во второй раз, так как шифрование происходит при помощи обычного гаммирования. И если криптоаналитик получит два сообщения, зашифрованных одним и тем же ключом, то он сможет на основе этого выявить сам ключ и впоследствии расшифровать сообщения.

Таким образом, из-за практичной сложности использования абсолютно криптостойких алгоритмов, часто используют именно вычислительно криптостойкие.

5. Симметричные алгоритмы

В классической криптографии выделяют два метода шифрования:
1. Подстановка (Substitution)
2. Перестановка (Permutation)

- К подстаночным шифрам причисляют следующие типы:

1. Одноалфавитные шифры (шифр Цезаря, Аффинный шифр, шифр Атбаш, шифр Бэкона, шифр Полибия, шифр пар)

2. Полиалфавитные шифры (шифр Виженера, шифр Бофора, шифр Гронсфельда, шифр Вернама)

3. Вероятностные шифры (омофонический шифр)

4. Полиграммные шифры (шифр Хилла, шифр Плейфера, шифр Порте)

- К перестановочным шифрам причисляют следующие типы:

1. Шифры одинарной перестановки (шифр Древней Спарты, маршрутные шифры)
2. Шифры множественной перестановки (шифр двойной перестановки)

В современной криптографии подстановка и перестановка никогда не используются по отдельности, придерживаясь теории связи, выдвинутой Шенноном, где необходимо использовать диффузию (рассеивание) и конфузию (перемешивание) в алгоритмах.

В настоящее время используются два типа симметричных алгоритмов:
1. Блочные шифры – оперируют фиксированной длиной битов.
2. Потоковые шифры – оперируют последовательностью входных битов.

Блочные шифры чаще всего надёжнее потоковых, но шифруют данные медленнее.

В поточных шифрах используется такой термин, как гаммирование. Иными словами, это наложение битов ключа на открытое/закрытое сообщение под средством операции xor (исключающее или).
$E_K(M) = K_i$ xor M_i, где i – это индекс бита или байта.

Операция xor обладает свойстами коммутативности (a xor b = b xor a) и ассоциативности ((a xor b) xor c = a xor (b xor c)), тем самым D(E(M)) = E(D(M)) = M. Помимо этого, функции E и D полностью эквивалентны, тем самым E = D, соответственно E(E(M)) = M.

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0

M = 0101, K = 1100

C = $E_K(M) = M_i$ xor K_i = 0101 xor 1100 = 1001

M = $E_K(C) = E_K(E_K(M)) = C_i$ xor K_i = 1001 xor 1100 = 0101

6. Асимметричные алгоритмы

Асимметричная криптография очень тесно связана с математикой и работой с большими числами.

Асимметричные алгоритмы также называются алгоритмами с открытым ключом.

1. Революция в мире криптографии началась с момента создания протокола Диффи-Хеллмана, в котором решалась проблема обмена ключей в незащищённом от прослушивания канале связи.

Описание протокола следующее:
1. Алиса генерирует случайное простое число p, которое будет являться делителем по модулю.
2. Алиса генерирует случайное число x, которое будет являться первообразным корнем.

3. Алиса генерирует случайное натурально число a, которое будет являться секретным числом (закрытым ключом).
4. Алиса вычисляет значение A (открытый ключ) по следующей формуле:
$A = x^{\mathbf{a}} \bmod p$
5. Алиса передаёт Бобу значения p, x, A.
6. Боб генерирует случайное число b, которое будет являться секретным числом (закрытым ключом).
7. Боб вычисляет сеансовый ключ по следующей формуле:
$K = A^{\mathbf{b}} \bmod p$
8. Боб вычисляет значение B (открытый ключ) по следующей формуле:
$B = x^{\mathbf{b}} \bmod p$
9. Боб передаёт Алисе значение B.
10. Алиса вычисляет сеансовый ключ по следующей формуле:
$K = B^{\mathbf{a}} \bmod p$

Как итог, Алиса и Боб получили общий сеансовый ключ K, который они впоследствии могут применять для шифрования сообщений.

Простота числа p даёт однозначность решения. Возведение же в степень выглядит следующим образом:
$A^{\mathbf{b}} \bmod p = (x^{\mathbf{a}})^{\mathbf{b}} \bmod p = (x^{\mathbf{b}})^{\mathbf{a}} \bmod p = B^{\mathbf{a}} \bmod p = K$
В математическом описании этот протокол будет выглядить следующим образом:

1. [A] a, x, p, $A = x^{\mathbf{a}} \bmod p$
2. [A] → x, p, A → [B]
3. [B] b, $K = A^{\mathbf{b}} \bmod p$, $B = x^{\mathbf{b}} \bmod p$
4. [B] → B → [A]
5. [A] $K = B^{\mathbf{a}} \bmod p$

Для нахождения большого простого числа p, применяют чаще всего малую теорему Ферма:
$x^{\mathbf{p-1}} = 1 \pmod p$, где x – какое-либо число.
Данную формулу можно расписать немного иначе: $x^{\mathbf{p-1}} \bmod p = 1$

У данной теоремы есть некоторый минус, при котором число p может оказаться состаным числом, но такая вероятность достаточно мала.

Протокол Диффи-Хеллмана основан на сложности задачи дискретного логарифмирования, то-есть сложно найти такое число a, зная даже числа A, x и p.
$x^{\mathbf{a}} = A \pmod p$
$\log_{\mathbf{x}} A = a \pmod p$
Иными словами, всё сводится к перебору возможных значений числа a.

2. На основе протокола Диффи-Хеллмана был создан криптографический алгоритм Эль-Гамаля:

1. Боб генерирует простое число p.

2. Боб генерирует число x, которое является первообразным корнем.
3. Боб генерирует случайное натурально число b, которое будет являться секретным числом.
4. Боб вычисляет значение B (открытый ключ) по следующей формуле:
$B = x^b \bmod p$
5. Боб публикует значения p, x, B.
6. Алиса генерирует случайное число r и создаёт открытое сообщение M.
7. Алиса вычисляет C1 и C2 по следующим формулам:
$C1 = x^r \bmod p$
$C2 = M B^r \bmod p$
8. Алиса передаёт Бобу получившиеся C1 и C2.
9. Боб вычисляет открытое сообщение P по следующей формуле:
$M = C2 / C1^b = (M B^r \bmod p) / (x^r \bmod p)^b =$
$(M B^r \bmod p) / ((x^r)^b \bmod p) =$
$(M B^r \bmod p) / (B^r \bmod p)$

В математическом описании использование алгоритма Эль-Гамаля будет выглядить следующим образом:
1. [B] b, p, x, $B = x^b \bmod p$
2. [B] → p, x, B → [A]
3. [A] r, M, $C1 = x^r \bmod p$, $C2 = M B^r \bmod p$
4. [A] → C1, C2 → [B]
5. [B] $M = C2 / C1^b$

3. Помимо сложности задачи дискретного логарифмирования, существует задача факторизации целых чисел, на которой строится алгоритм RSA.

RSA использует функцию Эйлера, которая принимает в качестве параметра какое-либо число и возвращает количество взаимнопростых чисел.

Функция Эйлера является мультипликативной, иными словами f(qp) = f(q)f(p)

Если числа q и p являются простыми, то результат функции Эйлера будет следующим:
f(qp) = f(q)f(p) = (q-1)(p-1)

Алгоритм генерации ключей в RSA следующий:
1. Алиса генерирует два простых числа p и q.
2. Алиса вычисляет число n = p q.
3. Алиса вычисляет функцию Эйлера
f(n) = f(pq) = f(p)f(q) = (p-1)(q-1)
3.5. Уничтожение чисел p и q.
4. Выбрать случайным образом число e (открытый ключ), которое будет являться взаимнопростым к числу f(n).
5. Вычислить число d (закрытый ключ), которое будет удовлетворять следующему уравнению:
ed = 1 (mod f(n))
ed mod f(n) = 1
где e /= d, p < d < f(n) и q < d < f(n) иными словами число d должно быть обратным к числу e.

В математическом описании алгоритм генерации ключей RSA будет выглядить следующим образом:
1. [A] p, q, n = p q, f(n) = f(pq) = (p-1)(q-1)
2. [A] НОД(e, f(n)) = 1
3. [A] d, где ed = 1 (mod f(n)) и e /= d и p < d < f(n) и q < d < f(n)

Алгоритм шифрования RSA следующий:
1. Алиса публикует открытый ключ {e, n}
2. Боб создаёт открытое сообщение M
3. Боб вычисляет закрытое сообщение:
$C = M^e \bmod n$
4. Боб отправляет Алисе закрытое сообщение C.
5. Алиса вычисляет открытое сообщение:
$M = C^d \bmod n$

В математическом описании использование алгоритма шифрования RSA будет выглядить следующим образом:
1. [A] → {e, n} → [B]
2. [B] M, $C = M^e \bmod n$
3. [B] → C → [A]
4. [A] $M = C^d \bmod n$

4. До создания методов шифрования RSA и Эль-Гамаля был предложен метод под названием ранцевый алгоритм, где чаще всего фигурирует схема Меркла-Хеллмана (алгоритм некриптостойкий).
Генерация ключей в алгоритме Меркла-Хеллмана:
1. Алиса создаёт сверхвозрастающую последовательность - закрытый ключ

(последовательность чисел, в которой каждое следующее число больше суммы всех предыдущих чисел).
Закрытый ключ: {2, 3, 6, 13, 27, 52}
2. Далее Алиса вычисляет нормальную последовательность - открытый ключ. Для создания открытого ключа необходимо два числа (x,p) и закрытый ключ. Число p – это значение модуля, которое должно быть больше суммы всех чисел закрытого ключа, допустим число 105.
Число x должно быть взаимно простым с числом p, возьмём число 31. Вычисляется открытый ключ следующим образом:
2 * 31 mod 105 = 62
3 * 31 mod 105 = 93
6 * 31 mod 105 = 81
13 * 31 mod 105 = 88
27 * 31 mod 105 = 102
52 * 31 mod 105 = 37
Открытый ключ: {62, 93, 81, 88, 102, 37}

Математическое описание генерации ключей в алгоритме Меркла-Хеллмана:
1. [A] D_A, x, p, E_A = {D_{Ai} x mod p}

Алгоритм шифрования Меркла-хеллмана следующий:

1. Боб имеет какое-либо открытое сообщение и переводит его в двоичную форму деля его при этом на

блоки, равные количеству элементов в открытом тексте и берёт открытый ключ Алисы.
Открытое сообщение: 0110001101011101110
Открытый ключ: {62, 93, 81, 88, 102, 37}
Разбитое открытое сообщение:
011000 - 110101 – 101110
2. Шифрование строится на сложении чисел открытого ключа по открытому тексту таким образом, где 0 – это отсуствие числа, а 1 – его присутствие.
Первый блок: 93 + 81 = 174
Второй блок: 62 + 93 + 88 + 37 = 280
Третий блок: 62 + 81 + 88 + 102 = 333
Шифротекст: 174, 280, 333
3. Боб передаёт Алисе зашифрованное сообщение.
4. Чтобы Алисе расшифровать сообщение, ей необходимо будет найти обратное число числу x по модулю на p. Иными словами вычислить число y, где x
$y = 1 \pmod p$
Для вычисления обратного числа чаще всего применяют расширенный алгоритм Евклида. Тем не менее, можно решать подобные задачи при помощи малой теоремы Ферма:
$x^{p-1} = 1 \pmod p$
$x^{p-1} = x^{p-2}\, x \pmod p$
$x^{p-2} = y \pmod p$
но данные вычисления в нашем случае не помогут, так как малая теорема Ферма работает только тогда, когда число p является простым. Расширенный алгоритм Евклида же работает не только с простыми числами, но

и с взаимнопростыми. [Алгоритм Евклида и малая теорема Ферма разобраны в главе 3, раздел 1].
Обратное число 31 по модулю 105 равняется числу 61.
Теперь зная обратное число, достаточно легко вычислить открытое сообщение по закрытому ключу:
Закрытый ключ: {2, 3, 6, 13, 27, 52}
174 * 61 mod 105 = 9 = 3 + 6 = 011000
280 * 61 mod 105 = 70 = 2 + 3 + 13 + 52 = 110101
333 * 61 mod 105 = 48 = 2 + 6 + 13 + 27 = 101110

Математическое описание шифрования алгоритма Меркла-Хеллмана следующее:

1. [B] M, C = $E_A(M) \rightarrow C \rightarrow$ [A], где
$E_A = f(M, K) =$

$$\sum_{i=0}^{len(K)} \begin{cases} 0, \text{ если } M_i = 0 \\ K_i, \text{ если } M_i = 1 \end{cases}$$

где K – нормальная последовательность чисел открытого ключа,
f – функция сложения чисел из множества K.

2. [A] y, p, M = $D_A(C)$
$D_A = f(Q, K) =$

$$\sum_{i=len(K)}^{0} \begin{cases} 0, \text{ если } Q < K_i \\ 1, \text{ если } Q >= K_i,\ Q = Q - K_i \end{cases}$$

где $Q = C^y \bmod p$,

K – сверхвозрастающая последовательность чисел закрытого ключа,
f – функция разложения числа на сумму чисел из множества K.

Хоть мы и разобрали данный метод шифрования, тем не менее он не является надёжным. Криптосистема Меркла-Хеллмана была взломана, так как существует большая вероятность восстановить сверхвозрастающую последовательность по нормальной последовательности. Все методы шифрования основанные на ранцевом алгоритме также оказались уязвимы и в настоящее время они не используются.

Существует ещё множество алгоритмов с открытым ключом, тем не менее, все имеющиеся основаны либо на сложности дискретного логарифмирования, либо на сложности факторизации, либо на сложности укладки ранца.

7. Протоколы

Протоколы описывают последовательность действий, где учавствуют два или более лиц, для решения какой-либо проблемы.

Общее правило протоколов можно сформулировать следующим образом:
- Невозможно сделать или узнать больше, чем предусмотрено протоколом.
Криптографические протоколы описывают проблему и её решение с использованием криптографических алгоритмов. Они чаще всего затрагивают такие проблемы, как обмен ключами, использование цифровых подписей, аутентификация, невозможность отказа от авторства, тайное голосование, жеребьёвка и тд.

I. *Обмен ключами*

Как обменяться ключами таким образом, чтобы третья сторона никак не смогла узнать сеансовый ключ?

Ранее мы рассматривали протокол Диффи-Хеллмана, который предусматривал обмен ключами в незащищённом от прослушивания канале связи. Тем не менее до представления данного протокола, была иная идея, в которой использовалась криптография с открытым ключом, но она была скорее теоретической,

нежели практической. Её назвали головоломками Меркла.
Протокол следующий:
1. Боб создаёт около миллиона сообщений, в которых указывает два значения x и y, где x – это номер сообщения, а y – это закрытый ключ (x и y не должны повторяться). Боб сохраняет открытые копии в незашифрованном виде (для себя). Далее он шифрует все сообщения симметричным алгоритмом со случайным ключом и выставляет их в открытый доступ.
2. Алиса выбирает случайным образом одно из миллиона зишифрованных сообщений и применяет на это сообщение лобовую атаку (брутфорс).
3. Получив значения x и y, Алиса шифрует своё сообщение ключом y и отсылает зашифрованное сообщение вместе с числом x.
4. Боб получив зашифрованное сообщение и число x, может легко найти значение y из своих сообщений и впоследствии расшифровать сообщение Алисы.

Математическое описание головоломок Меркла:
1. [B] $P_2^{20} = \{x, y\}$, $C_2^{20} = E_R(P_2^{20})$
2. [A] $D_R(C_i) = P_i = \{x, y\}$
3. [A] M, $C = E_y(M) \rightarrow C, x \rightarrow$ [B]
4. [B] x:y, $M = D_y(C)$

Сложность для Евы состоит в том, что ей необходимо применить лобовую атаку для всех сообщений, пока не найдёт число x, которое передавала Алиса. Алисе же

следует применить лобовую атаку только для одного сообщения.

Помимо протокола Диффи-Хеллмана также часто применяют протоколы основанные на ассиметричных шифрах для обмена сеансовыми ключами:
1. Алиса генерирует приватный и публичный ключи (как пример, при помощи RSA).
2. Алиса отправляет Бобу свой публичный ключ.
3. Боб генерирует сеансовый ключ и зашифровывает его публичным ключом Алисы. Далее отпправляет зашифрованный ключ Алисе.
4. Алиса расшифровывает сеансовый ключ Боба при помощи своего закрытого ключа.

Математическое описание протокола обмена ключей:
1. [A] E_A, $D_A \rightarrow E_A \rightarrow$ [B]
2. [B] K, $C = E_A(K) \rightarrow C \rightarrow$ [A]
3. [A] $D_A(C) = K$

Генерация ключа также может проводиться и на стороне Алисы. Соответственно она должна узнать открытый ключ Боба и впоследствии зашифровать сеансовый ключ его открытым ключом.

Стоит заметить, что Ева в данном случае совершенно не имеет возможностей узнать сеансовый ключ, тем не менее, Мэлори способен подменить открытый ключ Алисы на свой публичный ключ. Когда Боб будет отправлять зашифрованный сеансовый ключ, Мэлори

будет способен его расшифровать, узнать сам ключ, а потом зашифровать его при помощи открытого ключа Алисы и передать ей. Как итог, Мэлори будет исполнять роль человека по середине, способного как читать, так и изменять сообщения.
Данная атака очень болезнена, так как избавиться от неё в одноранговых системах не представляется возможным из-за отсутствия центра распределения ключей (ЦРК), который будет гарантированно предоставлять открытый ключ нужного человека подтверждённым своим закрытым ключом в качестве цифровой подписи. Но никто не запрещает стать “сервером” в одноранговой сети, который будет исполнять роль ЦРК.

II. *Цифровая подпись*

Как доказать, что сообщение не подвергалось изменениям? Как доказать, что именно это сообщение отправили вы, а не кто-то другой?

Чтобы решить данные проблемы применяют цифровые подписи. Для получения цифровой подписи применяются обратные действия в асимметричных алгоритмах. Подписание сообщения – это функция D (закрытый ключ), проверка подписи – это функция E (открытый ключ).

Разберём самый простой алгоритм с цифровой подписью:

1. Алиса генерирует приватный и публичный ключи, а также уже имеет какое-либо сообщение. Ставит на него цифровую подпись, при помощи своего закрытого ключа.
2. Алиса посылает Бобу открытое сообщение и подписанное сообщение.
3. Боб берёт открытый ключ Алисы и расшифровывает полученное подписанное сообщение.
4. Сравнивает расшифрованное сообщение с открытым сообщением.

В математическом описании это будет выглядить следующим образом:
1. [A] E_A, D_A, M, S = D_A(M) → E_A, S, M → [B]
2. [B] M’ = E_A(S), M’ = M

Стоит упомянуть, что данное описание не является применимым на практике, так как цифровую подпись применяют не на всё сообщение, а на результат криптографической хеш-функции.
Соответственно, немного изменим математическое описание, добавив функцию H:
1. [A] E_A, D_A, M, S = D_A(H(M)) → E_A, S, M → [B]
2. [B] M’ = E_A(S), M’ = H(M)

Даже сейчас данное описание не является нормальным протоколом, где применяется цифровая подпись, так как само сообщение M никак не шифруется. Тем самым, нам необходимо добавить функцию шифрования с

сеансовым ключом. Протоколом предполагается, что уже был произведён обмен сеансовыми ключами.
1. [A] M, E_A, D_A, C = E_K(M), S = D_A(H(M)) → E_A, S, C → [B]
2. [B] M = D_K(C), M’ = E_A(S), M’ = H(M)

Стоит заметить, что подпись применяется именно на открытое сообщение, как и в реальной жизни.

III. *Аутентификация*

Как узнать, что именно вы зашли в систему, а не кто-либо другой?

Существует протокол, где применяется аутентификация с односторонними хеш-функциями:
1. Алиса посылает серверу свой пароль.
2. Сервер имеет в базе данных хешированный пароль.
3. Сервер хеширует пароль отправленный Алисой и сравнивает его с хешированным паролем хранящимся в базе данных.
Математическое описание:
1. [A] P → P → [B]
2. [B] C, H(P) = C

В данном случае существует некая проблема. Что если Ева получит каким-то образом хеш-значения пользователей? Это достаточно плохо, так как атака по словарю очень эффективна. В этом случае, чаще всего, сервер конкатенирует пароль с солью (случайной

строкой в несколько байт), далее вычисляет хеш-значение получившегося пароля и хранит его вместе с солью.
Сложность для Евы заключается в том, что ей необходимо будет подбирать вместе с паролем несколько байт случайной строки. Соответственно, атака по словарю будет уже менее эффективна.
Математическое описание аутентификации с солью:
1. [A] P → P → [B]
2. [B] C, R, H(P + R) = C, где R – соль

В вышеприведённом протоколе существует некая проблема, что если пароль передаётся в открытом виде (используется HTTP вместо HTTPS)? Что если Ева находится у сервера и способна читать пароли до их хеширования? Чтобы избавиться от этих проблем, может применяться другой протокол аутентификации, который основан уже на асимметричной криптографии:
1. Алиса отправляет серверу своё имя и запрос на получение случайной строки.
2. Сервер получает имя Алисы и выявляет в базе данных её открытый ключ. Далее он отправляет Алисе случайно сгенерированную строку.
3. Алиса зашифровывает строку при помощи своего закрытого ключа (цифровая подпись) и отправляет серверу.
4. Сервер расшифровывает сообщение отправленное Алисой при помощи её открытого ключа и сравнивает со сгенерированной строкой, на первом этапе.

Математическое описание протокола аутентификации с открытым ключом следующее:
1. [A] → Alice → [B]
2. [B] E_A, R → R → [A]
3. [A] C = D_A(R) → C → [B]
4. [B] R’ = E_A(C), R’ = R

IV. *Аутентификация и обмен ключами*

Является объединением задач протоколов для защиты от атак “человек посередине” (MITM).

В данных протоколах присутствует такая личность как Трент, иными словами доверенное лицо, с которым Алиса или Боб используют совместные ключи для шифрования сообщений.

Подобных протоколов достаточно много, так что были выбраны одни из самых интересных и эффективных.

- Протокол Ньюмана-Стаблбайна:

1. Алиса генерирует случайное число. Отправляет Бобу своё имя и случайноое число.
2. Боб генерирует случайное число. Устанавливает метку времени по своим часам. Зашифровывает имя Алисы + её случайное число + метку времени ключом, который используется совместно с Трентом.
3. Посылает Тренту своё имя, своё случайное число и зашифрованное сообщение.

4. Трент генерирует сеансовый ключ. Зашифровывает общим ключом Алисы имя Боба + случайное число Алисы + сеансовый ключ и + метку времени Боба. Зашифровывает общим ключом Боба имя Алисы + сеансовый ключ + метку времени Боба. Передаёт Алисе оба зашифрованных сообщения и случайное число Боба.
5. Алиса расшифровывает сообщение Трента. Убеждается что связь с Бобом. Получает сеансовый ключ, сравнивает случайное число со своим случайным числом. Алиса шифрует случайное число Боба сеансовым ключом. Алиса отправляет Бобу сообщение Трента и зашифрованное случайное число Боба.
6. Боб расшифровывает сообщение Трента. Убеждается что связь с Алисой. Получает сеансовый ключ, метку времени. Проверяет метку времени. Далее расшифровывает случайное число и сравнивает его со своим числом.

Математическое описание протокола Ньюмана-Стаблбайна:
1. [A] $R_{\mathbf{A}} \rightarrow$ Alice, $R_{\mathbf{A}} \rightarrow$ [B]
2. [B] $R_{\mathbf{B}}$, $T_{\mathbf{B}}$, $C_{\mathbf{B}} = E_{\mathbf{BT}}(\text{Alice}, R_{\mathbf{A}}, T_{\mathbf{B}}) \rightarrow \text{Bob}, R_{\mathbf{B}}, C_{\mathbf{B}} \rightarrow$ [T]
3. [T] $D_{\mathbf{BT}}(C_{\mathbf{B}})$, K, $C_{\mathbf{A}} = E_{\mathbf{AT}}(\text{Bob}, R_{\mathbf{A}}, K, T_{\mathbf{B}}), C_{\mathbf{B}} = E_{\mathbf{BT}}(\text{Alice}, K, T_{\mathbf{B}}) \rightarrow C_{\mathbf{A}}, C_{\mathbf{B}}, R_{\mathbf{B}} \rightarrow$ [A]
4. [A] $D_{\mathbf{AT}}(C_{\mathbf{A}})$, $C = E_{\mathbf{K}}(R_{\mathbf{B}}) \rightarrow C_{\mathbf{B}}, C \rightarrow$ [B]
5. [B] $D_{\mathbf{BT}}(C_{\mathbf{B}})$, $D_{\mathbf{K}}(C)$

У данного протокола есть дополнительная особенность, при которой последующая аутентификация не требует действий со стороны Трента.

1. Алиса генерирует новое случайное число. Посылает Бобу зашифрованное сообщение Трентом на этапе 5 вместе с новым случайным числом.
2. Боб генерирует новое случайное число. Зашифровывает сеансовым ключом случайное число Алисы. И отправляет это всё Алисе.
3. Алиса зашифровывает случайное число Боба сеансовым ключом и отправляет ему.

Математическое описание повторной аутентификации в протоколе Ньюмана-Стаблбайна:
1. [A] R_A', $C_B = E_{BT}(Alice, K, T_B) \rightarrow R_A'$, $C_B \rightarrow [B]$
2. [B] $D_{BT}(C_B)$, R_B', $C = E_K(R_A') \rightarrow R_B'$, $C \rightarrow [A]$
3. [A] $D_K(C)$, $C = E_K(R_B') \rightarrow C \rightarrow [B]$

Новые случайные числа предотвращают атаки с повторной передачей сообщений.

- Протокол Деннинга-Сакко:

1. Алиса отправляет Тренту своё имя и имя Боба.
2. Трент отправляет Алисе имя Боба + открытый ключ Боба подписанные закрытым ключом Трента. Трент отправляет Алисе имя Алисы + открытый ключ Алисы подписанные закрытым ключом Трента.

3. Алиса генерирует сеансовый ключ, создаёт метку времени, далее подписывает своим закрытым ключом своё имя + имя Боба + сеансовый ключ + метку времени и зашифровывает данную подпись открытым ключом Боба.
4. Алиса отправляет Бобу сообщение предыдущего этапа и два сообщения Трента.
5. Боб расшифровывает сообщение Алисы при помощи своего закрытого ключа. Проверяет подпись Алисы с помощью её открытого ключа. Проверяет имена сеанса связи. Проверяет корректность метки времени.

Математическое описание протокола Деннинга-Сакко:
1. [A] → Alice, Bob → [T]
2. [T] $S_1 = D_T(Alice, E_A)$, $S_2 = D_T(Bob, E_B)$ → S_1, S_2 → [A]
3. [A] $E_T(S_1)$, $E_T(S_2)$, K, T_A, $C = E_B(D_A(Alice, Bob, K, T_A))$ → C, S_1, S_2 → [B]
4. [B] $D_B(E_A(C))$, $E_T(S_1)$, $E_T(S_2)$

V. *Разбиение секрета*

Как спроектировать такую систему, где один пользователь без другого не способен узнать открытое сообщение?

В данном случае самый простой способ разбиения секрета – это использование одноразового блокнота. Вы генерируете несколько случайных последовательностей равных длине сообщения, применяете гаммирование и на выходе получаете закрытое сообщение. Чтобы

узнать открытое сообщение, также необходимо произвести гаммирование ключами, но уже на закрытое сообщение.
Трент является тем, кто знает сообщение M.
Зашифрованное сообщение C может знать любой.
1. [T] M, K_1, K_2, C = M xor K_1 xor K_2
2. [T] → K_1 → [A]
3. [T] → K_2 → [B]

Чтобы получить сообщение M, Алиса и Боб должны скооперироваться и применить свои ключи на зашифрованное сообщение C:
[A][B] C, K_1, K_2, M = C xor K_1 xor K_2

В данном протоколе есть некоторые замечания. Например, что если Алиса и Боб смогут скооперироваться, когда это было не выгодно Тренту, и как итог узнать секрет намного раньше?
В данном случае можно предпринять меру, где увеличивается количество участников протокола. Но, что если один из участников протокола куда-либо пропадёт? После этого секрет останется секретом навсегда. Соответственно представленный протокол ещё больше ухудшается, когда количество участников становится намного больше.
При таких обстоятельствах применяется протокол с использованием уровней доступа. Как пример, усовершенствуем вышеприведённый протокол, чтобы Трент смог легко расшифровать сообщение в одиночку.

Но если Алиса захочет расшифровать сообщение, то ей необходимо будет объединиться с Бобом:
1. [T] M, K_1, K_2, K_3 = K_1 xor K_2, C = M xor K_3
2. [T] → K_1 → [A]
3. [T] → K_2 → [B]

[A][B] C, K_1, K_2, M = C xor K_1 xor K_2
[T] C, K_3, M = C xor K_3

Таким образом, при помощи гаммирования ключей, мы можем увеличивать уровень доступа и выстраивать определённые иерархии, где несколько единиц директоров способны заменить десяток бухгалтеров и сотни рядовых рабочих.

8. Дополнительная литература

“Взломщики кодов” (Д. Кан) – история криптографии.
“Книга шифров. Тайная история шифров и их расшифровки” (С. Сингх) – классическая криптография.
“Прикладная криптография. Протоколы, алгоритмы и исходные коды на языке С” (Б. Шнайер) – современная криптография.
“Криптография” (А. Молдовян, Н. Молдовян, Б. Советов) - современная криптография.

II. Golang

Если посмотреть на авторов данного языка программирования, какая компания его поддерживает и какие особенности есть у данного языка, то его популярность остаётся лишь вопросом времени.

Компания: Google.

Главные разработчики языка:
Кен Томпсон – один из главных разработчиков операционной системы UNIX и языка программирования Си.
Роб Пайк – один из разработчиков операционной системы UNIX и создатель кодировки UTF-8.

Помимо прямых разработчиков языка, также учавствовал в развитии и Брайан Керниган, который знаменит тем, что писал такие книги как “Язык программирования Си” в соавторстве с создателем языка Деннисом Ритчи, “UNIX. Программное окружение” и “Практика программирования” в соавторстве с Робом Пайком.

Помимо вышеперечисленного, была также написана книга “Язык программирования Go” в соавторстве с Аланом Донованом (одним из разработчиков языка Go).

1. Введение

Go в настоящее время ещё не слишком используемый язык для масштабной разработки больших проектов, тем не менее, очень прогрессивный и имеет достаточно много плюсов по сравнению с другими языками.

Чаще всего Go сравнивают по возможностям с языком Python, так как имеет также огромное количество стандартных библиотек, на нём также быстро можно писать программы и его можно использовать совершенно в разных областях. По-моему мнению, Go проигрывает Python’у лишь в количестве нестандартных библиотек, во всём остальном либо его превосходит, либо находится на одном уровне с ним.

Перед началом изучения данного языка, стоит разобрать его плюсы и минусы. Начнём пожалуй с минусов:

1. Код преимущественно в процедурной парадигме. Нет как таковой ООП (Объектно-ориентированной парадигмы) и ФП (Функциональной парадигмы). Хотя на счёт функциональной парадигмы можно от части поспорить. Если взять языки программирования Python, C++ и прочие, то можно увидеть, что они мультипарадигменные и содержат в себе ФП. Если смотреть с такой точки зрения, то Go также должен являться языком мультипарадигменным. Но в отличии от Python’а и C++ авторы языка Go не стали приписывать галочку, что язык поддерживает

функциональную парадигму и в этом они полностью правы, так как ни Python, ни C++ вовсе не способны использовать ФП. Всё что они считают за элементы такой парадигмы – это возврат функций и передача функций в качестве аргумента другим функциям (даже в языке программирования Си это уже было), но не поддерживают такие важные вещи в функциональных языках как ленивые вычисления и хвостовую рекурсию. Поэтому собственно ни язык программирования Go, ни языки программирования Java, C++, Python не являются функциональными.

2. Нестандартные идеи синтаксиса, обработки ошибок и конструирования пакетов. Этот минус действительно часто отпугивает программистов, которые привыкли писать с конструкциями try/catch, а также использовать static/extern для видимости областей. В Go перехват ошибок – это возврат значений из функций, точно также, как это было в языке программирования Си. Если вы хотите сделать переменную или функцию глобальной для других пакетов, то она должна начинаться с заглавной буквы, если же хотите сделать локальной переменную или функцию, то она должна начинаться с прописной буквы. Помимо таких небольших странностей, существуют определённые сценарии выборки компиляции пакетов. Допустим, если имя определённого файла в пакете заканчивается _linux.go или _windows.go, то компиляция данных пакетов будет происходить толькой под данной операционной системой.

3. “Деревянный” синтаксис. Если вы когда-либо программировали на языке Си, то скорее всего знаете силу указателей, знаете, что присваивание является также выражением, а скобки можно ставить в любом месте, хоть после функции через пробел, хоть на новой строке. В Go же это всё будет являться либо ошибкой синтаксиса, либо невозможностью использования таких операций над указателями.

Несмотря на вышеперечисленные минусы, Go обладает куда большим рядом достоинств:

1. Кроссплатформенный и кросскомпилируемый. Кроссплатформенностью языка уже вряд-ли кого-то из программистов можно удивить, но кросскомпилируемостью ещё как можно. Не каждый компилятор языка даёт возможность компилировать программу из под одной операционной системы для другой не переходя между ними.

Список поддерживаемых платформ:

Linux	linux
MacOS X	darwin
Windows	windows
FreeBSD	freebsd
NetBSD	netbsd
OpenBSD	openbsd
DragonFly BSD	dragonfly
Plan 9	plan9
Native Client	nacl
Android	android

Вместе с платформами можно также указывать желаемую архитектуру:

x386	386
AMD64	amd64
AMD64 (32 ptr)	amd64p32
ARM	arm

Теперь пример, допустим я нахожусь под операционной системой GNU/Linux с архитектурой amd64 и хочу скомпилировать программу для операционной системы Windows с архитектурой x386. Это всё будет выглядить следующим образом:

```
$ GOOS=windows GOARCH=386 go build main.go
```

2. Многопоточный. В Golang’е существует возможность использовать параллельные вычисления “из коробки” без подключения всеразличных библиотек. Само использование параллельности очень лёгкое, без всяких дополнительных функций, достаточно лишь указать ключевое слово go и после этого имя функции, которая должна выполняться параллельно. Чтобы управлять синхронизацией параллельных функций существуют chan’ы, а также пакет sync.

3. Огромное количество стандартных библиотек. Если сравнивать с языком Python, то скорее всего даже превосходит его.

4. Код легкочитаемый. Недостатки преимущественно процедурной парадигмы, а также “деревянности” языка могут стать даже его достоинством. Вся суть в том, что

язык достаточно прост со стороны малого количества ключевых слов, самой процедурной парадигмы, а также строго синтаксиса со стороны компилятора и стандартов языка по именованиям функций и методов. Всё это ведёт к тому, что даже новичок способен будет понять код программиста с многолетним стажем.

5. Быстрое написание кода. Go достаточно эффективен в разработке из-за возможности быстрого написания кода, так как обладает достаточно хорошей степенью абстракции в самом стандарте, при котором используются интерфейсы всеразличных методов.

На данный момент Go чаще всего используют как язык для написания серверной части веб-сайтов. Тем не менее, его также можно легко использовать для написания ПО, в том числе и с графическим интерфейсом (библиотеки gotk3, therecipe/qt, zserge/webview). На Go также очень легко работать с сетями на уровне транспортных протоколов TCP/UDP.

2. Начало

Синтаксис языка Go во многом унаследован от языка программирования Си.

Простая программа, которая выводит в терминал текст “hello, world”:

```
package main

import (
	"fmt"
)

func main() {
	fmt.Println("hello, world")
}
```

В этом примере есть несколько основных моментов, на которые стоит обратить внимание.

Во-первых, именование package main обязательно должно существовать в каждой программе как минимум один раз, так как пакет main – это главный/вызывающий и инициализирующий пакет.

Во-вторых, import указывает на то, какие пакеты мы хотим подключить. В данном случае пакет fmt отвечает за ввод и вывод текстовой информации.

В-третьих, функция main является главной функцией, которая должна встречаться всегда в целостной программе и вызываться ровно один раз.

Перед запуском программы предполагается, что компилятор уже установлен. Скачать его вы можете с оффициального сайта: golang.org

Чтобы скомпилировать программу, следует прописать в терминале следующее:
$ go build main.go

И чтобы запустить:
$ ./main

Знак ‘$’ указывает на то, что мы находимся в терминале.

3. Типы данных

Так как Go является строго типизированным языком, то без знания определённых типов данных нам не обойтись.

Основополагающие типы следующие:
I. Целочисленные типы:
1. int, int8, int16, int32, int64
   uint, uint8, uint16, uint32, uint64

где int – это знаковые числа,
   uint – это беззнаковые числа.

Число, которое стоит после int/uint – это количество занимаемых бит в памяти компьютера. У каждого числа есть определённый диапозон значений, допустим у числа uint8 = 2^{8} , где диапозон значений равен [0;255], у числа же int8 = 2^{8} , диапозон равен [-128;127].

Числа же int и uint без определённого количества бит варьируется от системы. int может занимать как 32 бита, так и 64 бита.

2. byte, rune

byte и rune являются синонимами чисел uint8 и int32.

byte = uint8
rune = int32

Они чаще всего, представляют собой хранение информации в виде массива байтов или рун.

II. Вещественные типы:
float32, float64

В отличии от целочисленных типов, вещественные не имеют варьированного количества бит занимаемой системой. Таким образом, нужно самому указывать сколько бит стоит потратить на вещественный тип.

III. Логический тип:
bool – имеет всего два значения: true или false.

IV. Строки:
string – тип, который хранит последовательность байтов. Представляет собой неизменяемую последовательность, в отличии от массивов и срезов байтов.

Объявление переменной в Go выглядит следующим образом:
var x int

где слева – это обозначение создания переменной, по-середине указывается имя переменной, а справа её тип.

Если же необходимо объявить сразу несколько переменных одного типа, то можно использовать следующую форму записи:
var x, y, z int

Если объявляется сразу несколько переменных, допустим разных типов, то можно использовать следующий метод объявления:

```
var (
	x int
	y float32
	z string
)
```

Стоит сказать, если мы не указали конкретное значение переменной, то автоматически к ней присваивается нулевое значение. Для типов int – это 0, для типов bool – это false, для типов string – это “”, для типов float – это 0.0.

Если необходимо инициализировать переменную, то вариантов больше:

```
var x int = 5
var x = 5
x := 5
```

Как видно, тип данных необязательно писать, когда инициализируется значение. Тем не менее, тип данных переменной x будет равняться int, а не int32, int64 и прочему. Таким образом, в данном случае стоит явно указать тип:

```
var x int32 = 5
var x = int32(5)
x := int32(5)
```

Не стоит путать знак := со знаком =, так как первый – это инициализация переменной, второй же – это присваивание переменной. Соответственно следующее будет корректным:

```
x := 5
x = 10
```

А это является ошибкой:

```
x := 5
x := 10
```

Существуют значения в программе, которые должны быть постоянными, иными словами служить константами. В Go на этот случай есть ключевое слово const, которое заменяет ключевое слово var:

```
const x int = 5
const x = 5
```

Если вы программировали на языке программирования C++, то const в языке Go равен constexp в C++. Иными словами, значение данной константы должно быть сразу вычисляемо до этапа компиляции. Допустим нельзя константе присвоить возвращаемое значение функции, тем не менее можно присвоить простое арифметическое выражение.

Помимо основных типов данных, в языке программирования Go присутствуют и составные: структуры, срезы, массивы и словари.

Начнём пожалуй с массивов, так как они являются самыми простыми из данного списка.
Их объявление будет выглядить следующим образом:
var array [5]int

Это говорит о том, что переменная array является массивом, который хранит 5 элементов типа int.

Чтобы обратиться к ячейке массива, следует указать её номер с нуля:
array[0] = 5
array[4] = 3

Инициализация массива выглядит следующим образом:
var array [5]int = [5]int{1,2,3,4,5}
var array = [5]int{1,2,3,4,5}
array := [5]int{1,2,3,4,5}

Можно также и не указывать количество элементов в массиве при инициализации, вместо этого можно указать три точки. В итоге компилятор сам вычислит количество элементов в массиве:
var array = [...]int{1,2,3,4,5}

Теперь переходим к срезам. Срезы объявляются схожим образом с массивами, но имеют динамическую длину:
var slice []int

Чтобы добавить элемент в срез, необходимо либо воспользоваться функцией append, которая расширяет срез, либо изначально инициализировать срез несколькими элементами при помощи функци make, а далее его заполнить:

```
var slice []int
slice = append(slice, 3, 5, 6)
```

или

```
var slice = make([]int, 3)
slice[0] = 3
slice[1] = 5
slice[2] = 6
```

Чтобы удалить из среза какой-либо элемент по индексу, стоит немного зайти вперёд и создать функцию removeInt на основе функции append:

```
func removeInt(slice []int, index int) []int {
	return append(slice[:index], slice[index+1:]...)
}
```

В данном примере, есть достаточно важные характеристики:

Во-первых, двоеточие в указании индекса – это диапозон, который мы берём.

Допустим, [:] - это все элементы среза.
[:index] – это все элементы до индекса не включая его.
[index:] – это все элементы после индекса включая его.

```
var slice = []int{2, 4, 6, 8, 10}
slice[:3] // Аналог – slice[0:3]
		// Результат – срез []int{2, 4, 6}
slice[3:] // Аналог – slice[3:5]
		// Результат – срез []int{8, 10}
slice[2:4] // Результат – срез []int{6, 8}
```

и так далее.

Во-вторых, троеточие после указания slice[index+1]. Так как функция append принимает в качестве второго аргумента неопределённое количество элементов, а не срез, то троеточие раскладывает список на элементы.

Теперь приступим к словарям. Словари также являются динамическими объектами. Соответственно, их легко можно удалить, их легко можно дополнять и изменять.

Объявление словарей выглядит следующим образом, где int – это ключ, string – это значение ключа:
var dict map[int]string

Для того, чтобы воспользоваться словарями, следует применить функцию make, которая предоставит место в памяти для его использования:
dict = make(map[int]string)

либо, можно сразу инициализировать словарь какими-либо значениями:

```
var dict = map[int]string{
	0: "Null",
	1: "One",
	2: "Two",
}
```

Дополнять словарь можно следующим образом:
dict[5] = "Five"

В данном случае не стоит путать словари со срезами, либо с массивами, потому что 5 в данном случае – это не индекс, а ключ, соответственно мы можем создать другой словарь, в котором ключ – это строка, а значение допустим тип bool:

```
var dict = map[string]bool {
	"First": true,
	"Second": true,
	"Third": false,
}
```

А дополнить можно следующим образом:
```
dict["Four"] = true
```

Стоит немного заострить внимание на том, что после каждого значения, в инициализации словаря, с новой строки обязательно нужно ставить запятую, даже на последнем элементе. Иначе, если этого не сделать, то компилятор Go заругается на синтаксическую ошибку.

Чтобы удалить элемент в словаре, стоит воспользоваться функцией delete:
```
var dict = map[string]bool {
	"First": true,
	"Second": true,
	"Third": false,
}
delete(dict, "Second")
```

В итоге, словарь будет выглядить следующим образом:
```
map[string]bool {
	"First": true,
	"Third": false,
}
```

И теперь, приступим к структурам. В языке программирования Go структуры являются очень важным элементом, так как они способны исполнять роль ООП.

Допустим создадим структуру “игрок”, у которого есть имя, количество жизней и наличие кота.

```
type player struct {
	name string
	health int
	cat bool
}
```

Структуры подобны классам, в которых могут содержаться как свойства, так и методы. В отличии от языка C++ методы создаются отдельно (не в поле структуры), тем не менее, создать метод в поле структуры вполне реально и это будет являться указателем на какую-либо функцию, подобно языку программирования Си:

```
type player struct {
	name string
	health int
	cat bool
	run func(p *player)	// метод run, принимающий
				// указатель на объект player
}
```

Но так лучше не делать, потому что при создании объекта, необходимо будет также указать и определённую функцию под это поле, к тому же этот метод не будет привязан к объекту.

Соответственно, имея структуру player:

```
type player struct {
	name string
	health int
	cat bool
}
```

Мы можем на основе её, создать объект:
```
var knight = player{"Tom", 100, true}
```

Также можно явно указать имена полей при инициализации:
```
var knight = player{
	name: "Tom",
	health: 100,
	cat: true,
}
```

Чтобы изменить поле в объекте, нужно указать имя объекта, поставить точку и после этого указать имя поля:
```
knight.health = 50
```

4. Операции с переменными

Всего существует достаточно мало операций и большинство связано с числами:

Оператор сложения: 1 + 3
Оператор вычитания: 5 – 2
Оператор умножения: 5 * 3
Оператор деления: 10 / 4
Оператор деления по модулю: 10 % 3

Также существуют операторы работы с битами:
Оператор ‘И’: 2 & 5
Оператор ‘ИЛИ’: 2 | 5
Оператор ‘НЕ’: ~5
Оператор ‘Сдвиг влево’: 1 << 5
Оператор ‘Сдвиг вправо’: 64 >> 1

Бинарные операторы также можно применять вместе с присваиванием:
x += 10 // прибавляет к x число 10
x *= 2 // умножает x на 2
и тд.

Существуют также унарные операторы инкремента и декремента:
x++ // увеличивает x на единицу
x-- // уменьшает x на единицу

Так как присваивание в языке программирования Go не является выражением, то невозможно сделать следующее:

```
y = x++ // Ошибка!
```

Также операторы инкремента и декремента возможны только в постфиксной записи, иными словами, нельзя записать их в префиксной форме:

```
// Ошибка!
++x
--x
```

Операторы сравнения (возвращают значения логического типа):
Оператор меньше: 5 < 3 // false
Оператор больше: 5 > 3 // true
Оператор меньше или равно: 5 <= 3 // false
Оператор больше или равно: 5 >= 3 // true
Оператор равно: 5 == 3 // false
Оператор не равно: 5 != 3 // true

Операторы работы с логическим типом:
Оператор ‘И’: true && false
Оператор ‘ИЛИ’: true || false
Оператор ‘НЕ’: !true

Не стоит путать операторы работы с битами и с логическим типом.

Стоит также уделить внимание конкатенации строк:
Оператор ‘+’: "hello, " + "world"

5. Условия

Условия в языке программирования Go выглядят следующим образом:

```
var flag bool
if !flag {
	flag = !flag
}
if flag {
	fmt.Println("true")
}
```

В данном случае, следует сказать три вещи. Первая - круглые скобки обхватывающие выражение необязательны. Вторая – фигурные скобки обязательно должны быть на одной линии с выражением, иначе синтаксическая ошибка:

```
// Ошибка
if !flag
{
	flag = !flag
}
```

```
// Верно
if !flag {
	flag = !flag
}
```

Третья – выражение обязательно должно быть логического типа. Допустим в языке Си нет как такового понятия логического типа, там всё – это числа. Соответственно может быть условие if (5 + 3 – 2) { /* какое-либо действие */ }. Тем не менее, в языке Go такое недопустимо.

6. Циклы

Всего в языке программирования Go выделено одно ключевое слово под циклы – это цикл for. Так или иначе, цикл for способен выполнять действия подобно классическому циклу for, циклу while, а также циклу foreach.

Классический вариант цикла for:

```
for i := 0; i < 10; i++ {
	fmt.Println(i)
}
```

Здесь стоит сказать, что нельзя ставить круглые скобки во все три выражения цикла:

```
// Ошибка!
for (i := 0; i < 10; i++) {
	fmt.Println(i)
}
```

Подобно классическому циклу for, мы можем откинуть некоторые части:

1.
```
var i = 0
for ;i < 10; i++ {
	fmt.Println(i)
}
```
2.
```
var i = 0
for ;i < 10; {
	fmt.Println(i)
	i++
}
```

Во втором варианте лучше отбросить символы ‘;’:
```
var i = 0
for i < 10 {
	fmt.Println(i)
	i++
}
```

Таким образом, мы получаем цикл while.

Циклами for/while можно сделать всё, но иногда удобнее воспользоваться циклом foreach, чтобы перебрать элементы какого-либо списка, массива или же словаря. В данном случае цикл for применяется вместе с ключевым словом range:

```
var slice = []int{2, 4, 6}

for index, value := range slice {
	fmt.Println(index, value)
}
// 0 2
// 1 4
// 2 6
var dict = map[string]bool {
	"First": true,
	"Second": true,
	"Third": false,
}

for key, value := range dict {
	fmt.Println(key, value)
}
// "First" true
// "Second" true
// "Third" false
```

Здесь всё хорошо работает, но если вам не понадобится допустим ключ, но он у вас будет указан, то компилятор Go заругается на то, что не используется переменная.

Эту ошибку можно исправить следующим образом:

```
for _, value := range dict {
	fmt.Println(value)
}
// true
// true
// false
```

Нижний знак подчёркивания означает специальную переменную в Go пространстве, которая служит заглушкой для неиспользуемых переменных.

Также, если вам нужно перебрать только ключи, то можно использовать два варианта:

1.
```
for key, _ := range dict {
	fmt.Println(key)
}
// "First"
// "Second"
// "Third"
```

2.
```
for key := range dict {
	fmt.Println(key)
}
```

```
// "First"
// "Second"
// "Third"
```

Так как вторая переменная необязательна, её можно опустить.

Если вам понадобится бесконечный цикл, то его можно реализовать несколькими способами:

1. for ;; { } // цикл for
2. for true { } // цикл while
3. for { } // цикл loop

Помимо самих циклов, с ними также часто применяются ключевые слова break, continue и куда реже goto.

```
// Выход из цикла
for i := 0; i < 7; i++ {
	if i == 5 {
		break
	}
	fmt.Println(i)
}
```

```
// 0
// 1
// 2
// 3
// 4

// Пропуск итерации цикла
for i := 0; i < 7; i++ {
	if i == 5 {
		continue
	}
	fmt.Println(i)
}
// 0
// 1
// 2
// 3
// 4
// 6
```

Если у вас есть вложенные циклы и необходимо, допустим, прервать все сразу, то может помочь оператор goto (не рекомендуется), либо же метки break и continue:

```
for i := 0; i < 3; i++ {
	for j := 0; j < 3; j++ {
		if i == 2 && j == 1 {
			goto end_loop
		}
		fmt.Println(i,j)
	}
}
end_loop:
fmt.Println("Continue code")
// 0 0
// 0 1
// 0 2
// 1 0
// 1 1
// 1 2
// 2 0
// Continue code
```

```
end_loop: for i := 0; i < 3; i++ {
	for j := 0; j < 3; j++ {
		if i == 2 && j == 1 {
			break end_loop
		}
		fmt.Println(i,j)
	}
}
fmt.Println("Continue code")
```

```
// 0 0
// 0 1
// 0 2
// 1 0
// 1 1
// 1 2
// 2 0
// Continue code
```

Если же необходимо пропустить какую-либо итерацию цикла, то также можно использовать ключевое слово continue с меткой:

```
pass_iter: for i := 0; i < 4; i++ {
	for j := 0; j < 4; j++ {
		if i == 2 && j == 1 {
			continue pass_iter
		}
		fmt.Println(i,j)
	}
}
fmt.Println("Continue code")
// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// 1 3
```

```
// 2 0
// 3 0
// 3 1
// 3 2
// 3 3
// Continue code
```

7. Функции

В языке программирования Go нет такого понятия как прототип функции, так как компилятор сразу просматривает все области видимости для их нахождения, в отличии от языка Си.

Таким образом, существует лишь определение функции и не имеет значения до или после функции main она была определена.

Синтаксис самой простой функции, которая ничего не принимает, ничего не возвращает и ничего не делает выглядит следующим образом:

```
func nothing() {
}
```

В круглых скобках указываются аргументы функции, после круглых скобок указываются возвращаемые значения, а внутри фигурных скобок – тело функции.

На основе этого, мы можем создать простую чистую функцию, которая складывает два целых числа:

```
func add(x int, y int) int {
	return x + y
}
```

Как итог, мы можем использовать эту функцию в своей программе следующим образом:

```
var (
	x = add(5, 6) // 11
	y = add(10, 20) // 30
	c = add(x, y) // 41
)
```

Запись ‘func add(x int, y int)’ можно немного упростить: ‘func add(x, y int)’, так как тип переменных вычисляется с правой стороны.

Функции также могут принимать указатель на какую-либо переменную для её изменения без возврата значения:

```
func inc(x *int) {
	*x++
}

var a = 10
inc(&a) // a = 11
```

Знак ‘*’ - является разыменованием адреса и получение по этому адресу значения переменной.
Знак ‘&’ - является взятием адреса переменной.

Впринципе сила указателей в Go на этом заканчивается. Нет никакой магии связанной с адресной арифметикой, залезанием в область памяти и тд, что присуще языку Си.

8. Методы

Методы очень схожи с функцией, за исключением того, что методы привязаны к созданному типу данных.

Самый простой пример – это создать синоним для целочисленного типа данных.

```
type myInt int32

func (x *myInt) add(y myInt) {
	*x += y
}

func main() {
	var z myInt = 10
	fmt.Println(z) // 10
	z.add(7)
	fmt.Println(z) // 17
}
```

Чаще всего методы применяются к объектам каких-либо структур. Как пример, возьмём прошлую структуру player:

```
type player struct {
	name string
	health int
	cat bool
}

 // лечение
func (p *player) treat(n int) {
	x := p.health + n
	if x > 100 {
		p.health = 100
		return
	}
	p.health = x
}

func main() {
	var knight = player{"Tom", 32, true}
	knight.treat(20)
	fmt.Println(knight.health) // 52
}
```

Заметьте, что указатель в языке Go может неявно опускаться при условии того, что мы работаем с объектом. В языке программирования Си, как пример,

для этого случая сделали специальный знак -> который позволяет работать с указателем на объект.

Хоть я и использовал примеры с указателями, тем не менее, с методами можно работать точно также, как и с обычными функциями.

9. Пакеты

Главное, что нужно знать при работе с пакетами – это понимание области видимости в языке Go, то-есть прописные буквы – локальная видимость, заглавные буквы – глобальная видимость.

Помимо этого нужно уточнить, что Go использует именно пакеты, а не модули. В итоге, мы работаем с директориями, а не с отдельными файлами .go.

Как пример, можно создать директорию utils рядом с главным файлом main.go. В директории utils создать пару файлов: add.go, sub.go.

```
// Файл: add.go
package utils

func Add(x, y int) {
	return x + y
}
```

```
// Файл: sub.go
package utils

func Sub(x, y int) {
	return x - y
}

// Файл: main.go
package main

import (
	"fmt"
	"./utils"
)

func main() {
	var (
		x = 5
		y = 10
		z int
	)
	z = utils.Add(x, y) // 15
	fmt.Println(z)
	fmt.Println(utils.Sub(z, x)) // 10
}
```

Указав директорию utils в текущей директории (./) мы можем импортировать все глобальные функции, методы, переменные и типы данных данного пакета.

Также мы можем изменить имя пакета, если допустим, это имя достаточно большое:

```
import (
	u"./utils"
)
```

и теперь можем обращаться к функциям следующим образом:
u.Add(x,y)
u.Sub(x,y)

Если же, необходимо вообще убрать именование пакета из вызовов функций, то перед импортированием пакета стоит указать точку.

```
import (
	."./utils"
)
```

и использовать теперь функции можно без имени пакета:
Add(x,y)
Sub(x,y)

Таким образом, пакеты предоставляют определённый уровень абстракции и инкапсуляции вместе со структурами и методами, тем самым играя роль ООП.

10. Параллельные вычисления

Часто возникает необходимость использовать параллельные вычисления в программе. Работают с ними программисты либо для увеличения производительности, либо для создания одновременно работающих процессов, например клиент и сервер на одной машине.

Функции, которые выполняют параллельные вычисления, в языке программирования Go называются горутинами. Также стоит сказать, что сама функция main тоже является горутиной. Соответственно, если вы создали горутину в main, то функция main может завершиться раньше, чем вызванная функция, тем самым мы не получим от неё результат.

Чтобы создать новую горутину, следует прописать ключевое слово go перед вызовом функции:

```
package main

import (
	"fmt"
)

func print_num(num int) {
	fmt.Print(num)
}
```

```
func main() {
	for i := 0; i < 5; i++ {
		go print_num(i)
	}
	print_num(5)
	fmt.Scanf("%d")
}
```

Функция Scanf нужна лишь для того, чтобы получить результат всех отработанных горутин. При нажатии enter - программа закроется.

Если вы запустите несколько раз данную программу, то результат часто будет отличаться:

```
// 142305
// 132450
// 510234
// . . .
```

Если вы уберёте функцию Scanf, то результат некоторых функций может даже не дойти до нас, так как main завершилась раньше:

```
// 145
// 142305
// 4215
// 5
```

Бывают ситуации, когда лучше сделать параллельную функцию, но при этом, чтобы в дальнейшем мы смогли использовать её результат без вероятности того, что так результат мы и не получим. На помощь прихотя каналы.

Каналы это составной тип данных, который выглядит следующим образом:
chan int
chan bool
chan string
и так далее.

Работа с каналами немного иная, чем с обычными переменными. В каналах для присваивания и передачи значения используется знак <- (стрелочка).

Пример использования следующий:

```
package main

import (
	"fmt"
)

func print_num(ch chan int) {
	var i int
	for i = 0; i < 3000000000; i++ {
	}
	ch <- i
}
```

```
func main() {
	var ch = make(chan int)
	go print_num(ch)

	fmt.Println("hello, world")
	// больше действий ...

	// ждём результат функции
	fmt.Println("Number:", ←ch)
	close(ch)
}

// hello, world
// Number: 3000000000
```

В данном примере, мы запускаем функцию print_num параллельно функции main. На промежутке от вызова функции и до ожидания результата, мы можем легко производить какие-либо действия в функции main не беспокоясь о том, что горутина решит вернуть результат не в подходящий момент, либо же вообще его не вернуть. Каналы позволяют синхронизировать горутины и получать от них ответы в нужном нам порядке.

Чтобы закрыть ненужный канал, следует использовать функцию close.

Если параллельная функция должна переодически передавать данные другой горутине, то можно воспользоваться циклом for, пока не закроется канал:

```
package main

import (
	"fmt"
)

func get_num(ch chan int) {
	for i := 0; i < 5; i++ {
		ch <- i
	}
	close(ch)
}

func main() {
	var ch = make(chan int)
	go get_num(ch)
	for num := range ch {
		fmt.Print(num)
	}
}

// 0123456789
```

Самая основная и часто встречаемая ошибка людей, пользующихся параллельными вычислениями – это использование глобальных переменных без их временной блокировки. Данные ошибки очень сложно найти и потому следует быть предельно аккуратными. Такой вид ошибок называют состоянием гонки, когда одновременно несколько горутин пытаются взаимодействовать с одним объектом.
Чтобы временно блокировать доступ к какому-либо участку кода, используют функции Lock/Unlock из пакета sync. Они предоставляют доступ только одной горутине и пока она не завершит своё действие над заблокированным полем, никакие другие горутины не смогут взаимодействовать с этим участком кода и будут находиться в очереди.

Пример использования функций Lock/Unlock:

```
package main

import (
	"fmt"
	"sync"
)

var GlobalInt int
var Mutex sync.Mutex
```

```
func plus() {
	Mutex.Lock()
	GlobalInt++
	Mutex.Unlock()
}

func minus() {
	Mutex.Lock()
	GlobalInt--
	Mutex.Unlock()
}

func main() {
	go minus()
	go plus()
	go plus()

	fmt.Scanf("%d")
	fmt.Println(GlobalInt)
}

// 1
```

Иногда бывает полезным определить группу горутин как единое целое, чтобы получить общий проделанный ими результат.

```
package main
```

```
import (
	"fmt"
	"sync"
)

func mul2(x *int, wg *sync.WaitGroup) {
	defer wg.Done()
	*x *= 2
}

func main() {
	var wg sync.WaitGroup
	var x = 1

	wg.Add(10)

	for i := 0; i < 10; i++ {
		go mul2(&x, &wg)
	}

	wg.Wait()
	fmt.Println(x)
}

// 1024
```

Ключевое слово defer выполняется только тогда, когда функция завершит своё выполнение.

Функция Add указывает на количество горутин в группе.

Функция Done сигнализирует о своём завершении.

Функция Wait ожидает ответа завершённости от каждой горутины.

11. Дополнительная литература

“Язык программирования Go” (Б. Керниган, А. Донован)
Сайт metanit с разделом Go:
https://metanit.com/go/tutorial/
https://metanit.com/go/web/

III. Практика

Данный раздел создан для объединения двух теоретических знаний в одно практическое. Будут разобраны детали на счёт сетевого программирования, математических алгоритмов и непосредственно пакета crypto.

1. Математические алгоритмы

1. *Малая теорема Ферма*

Одним из основных алгоритмов в криптографии с открытым ключом является малая теорема Ферма, в которой должно выполняться уравнение:

$a^{p} = a \pmod p$

где p – простое число.

Из этой формулы выходит два частных случая:

1. $a^{p-1} = 1 \pmod p$
2. $x^{p-2} = y \pmod p$

Первый случай помогает нам находить с большой вероятностью простые числа. Второй случай помогает узнать обратное число по модулю.

В малой теореме Ферма существует пара минусов. Во-первых, алгоритм является вероятностным, где существует возможность выбора составного числа, а также присутствует возможность выбора редких чисел,

которые по любому основанию b будут являться составными, но при этом результат теста Ферма будет положительным в сторону простого числа. Такие числа называют числами Кармайкла. Существует 255 чисел в диапозоне от нуля до 100 000 000. Несколько чисел Кармайкла: {561, 1105, 1729, 2465, 2821, 6601}. Во-вторых, алгоритм работает только с простым числом p. Иными словами, если число p взаимнопростое с числом a, но и в то же время p является составным, то алгоритм не будет работать.

Использование данного алгоритма предполагает работу с большими степенями числа. Обычное возведение очень ресурсозатрачиваемо, когда работа идёт со степенями, длина которых превосходит четыре-пять знака. Соответственно потребуется создать функцию, которая будет быстро возводить в степень число деля его при этом постоянно по модулю.

```
func expmod(x, y, z uint64) uint64 {
	var result, bit uint64
	result = 1
	for bit = 1; bit <= y; bit <<= 1 {
		if bit & y != 0 {
			result = (result * x) % z
		}
		x = (x * x) % z
	}
	return result
}
```

В данном случае алгоритм работает таким образом, что число y записывается сначала в двоичной форме счисления. Далее идёт перебор битов, если бит не равен нулю, тогда умножить число аккумулятор (result) на x и поделить по модулю на z. Далее к числу x присвоить квадрат x по модулю на z. После того, как число в переменной bit станет больше y, алгоритм прекратится и вернётся значение result.

```
// Малая теорема Ферма
func main() {
	var p uint64 = 684211457
	fmt.Println(expmod(5342, p, p)) // 5342
}

// Тест Ферма на простое число
func main() {
	var p uint64 = 684211457
	// 2 ^ p-1 mod p
	fmt.Println(expmod(2, p-1, p)) // 1
}

// Обратное число по модулю
func main() {
	var p uint64 = 684211457
	fmt.Println(expmod(5342, p-2, p)) // 663846496
}
```

2. *Алгоритм Евклида*

Данный алгоритм способен выявлять взаимнопростые числа. Если результат алгоритма (наибольший общий делитель) равен единице, то это свидетельствует о том, что оба числа взаимнопросты.

Алгоритм Евклида очень легко описывается со стороны рекурсии:
gcd(a, b) =
{ если b = 0, вернуть a
{ если b /= 0, тогда gcd(b, a mod b)

К тому же, если язык оптимизирован для хвостовой рекурсии, то вполне реально записать именно такой вариант. Но Go не поддерживает хвостовую рекурсию и соответственно со стороны оптимизации лучше всего воспользоваться циклами.

```
func gcd(a, b uint64) uint64 {
	for b != 0 {
		a, b = b, a % b
	}
	return a
}

// Наибольший общий делитель (НОД) = 178
func main() {
	fmt.Println(gcd(9847219831228, 827194853271 34))
}
```

3. *Функция Эйлера*

Данная функция вычисляет количество взаимнопростых чисел для числа n. Если значение функции Эйлера равно n-1 для числа n, то число n является простым.

Соответственно из данной функции можно выявить малую теорему Ферма.

Если f(n) = n-1,

тогда $a^{f(n)} = a^{n-1} = 1 \pmod{n}$

тогда $a^{f(n)-1} = a^{n-2} = x \pmod{n}$,

где x – обратное число a.

Также функция Эйлера является мультипликативной, иными словами:
f(n) = f(pq) = f(p)f(q), где n = pq.

```
func euler(num uint64) uint64 {
	var count uint64
	for i := uint64(1); i < num; i++ {
		if gcd(num, i) == 1 {
			count++
		}
	}
	return count
}
```

Теперь можно проверить работоспособность и правильность сделанной функции Эйлера:

```
fmt.Println(euler(101)) // 100
fmt.Println(euler(88)) // 40
fmt.Println(euler(8)*euler(11)) // 40
```

4. *Расширенный алгоритм Евклида.*

Данный алгоритм, помимо нахождения наибольшего общего делителя, также способен возвращать обратное число по модулю двух взаимнопростых чисел.

```
func extgcd(a, b int64) (int64, int64, int64) {
	var z, x1, x2, y1, y2 int64
	x1, x2, y1, y2 = 0, 1, 1, 0
	for b != 0 {
		z = a / b
		a, b = b, a - (z * b)
		x2, x1 = x1, x2 - (z * x1)
		y2, y1 = y1, y2 - (z * y1)
	}
	return a, x2, y2
}

func inverse(a, b int64) int64 {
	_, x, _ := extgcd(a, b)
	return (x % b) + b
}
```

```
// Обратное число: 663846496
func main() {
	fmt.Println(inverse(5342, 684211457))
}
```

Первый возвращаемый аргумент функцией ext_gcd – это наибольший общий делитель (НОД), второй и третий аргументы – это коэффициенты перед меньшим (x) и большим (y) числом:

ax + by = gcd(a,b)
если gcd(a,b) = 1, тогда упрощаем выражение
ax + by = 1
ax = 1 (mod b) – нахождение обратного числа x

Чтобы найти обратное число, необходимо взять коэффициент перед меньшим числом и сделать его положительным числом по модулю:
x = (x mod b) + b

2. Использование пакета crypto

Язык программирования Go имеет стандартный пакет для работы с криптографическими алгоритмами.

1. *Начнём пожалуй с функции генерирования сеансового ключа, которая использует пакет* ‘***crypto/rand***’*:*

```
func SessionKey(max int) []byte {
    var slice []byte = make([]byte, max)
    _, err := rand.Read(slice)
    if err != nil { return nil }
    for max = max - 1; max >= 0; max-- {
        slice[max] = slice[max] % 94 + 33
    }
    return slice
}
```

Изначально создаётся список указанного размера (в аргументах), далее он заполняется псевдослучайной последовательностью байт. Чтобы ключ не имел специальных и пустых символов, следует пробежаться по нему и делить по модулю на 94 прибавляя при этом 33. В итоге мы умещаемся в диапозон ASCII символов от 33 до 126 включительно.

Данную функцию уже можно использовать для генерации, как пример, своих временных паролей.
Но помимо этого, данную функцию можно использовать и для создания постоянных паролей.
Алгоритм следующий:

1. Сгенерировать примерно 1000000 символов и поместить их в новый файл.
2. Выбрать два случайных числа в диапозоне от нуля до миллиона, где первое число – это позиция символа в файле, а второе число – это количество символов отчитываемых от первого числа.

3. Запомнить два числа и сохранить файл.

Как итог, это даёт не только защиту вашему паролю от тех, кому удастся завладеть файлом, но и также вам куда легче будет запомнить два числа, нежели запоминать всю комбинацию символов.

2. *Переходим к другой функции, которая использует пакет ‘**crypto/sha256**’, то-есть криптографическую хеш-функцию:*

```
func HashSum(data []byte) []byte {
    var hashed = sha256.Sum256(data)
    return hashed[:]
}
```

В данном случае вычисляется сумма списка байтов и заносится в массив hashed. Чтобы вернуть не массив, а список, применяется двоеточие hashed[:].

Криптографическая хеш-функция применяться может как в проверке целостности документов, цифровой подписи, так и в сжимаемости публичного ключа.

3. *Генерация закрытого/открытого ключей. Используется два пакета: ‘**crypto/rsa**’ и ‘**crypto/rand**’.*

```
func GenerateKeys(bits int) (*rsa.PrivateKey,
*rsa.PublicKey) {
        priv, err := rsa.GenerateKey(rand.Reader, bits)
```

```
	utils.CheckError(err)
	return priv, &priv.PublicKey
}
```

Функция принимает битность ключа RSA. Далее используется функция GenerateKey, которая принимает Reader для случайно генерации ключа и количество бит. Проверяется наличие ошибок при помощи функции CheckError и возвращаются приватный/публичный ключи.

Пакет utils не стандартный, функция CheckError имеет следующий вид:

```
func CheckError(err error) {
	if err != nil {
		fmt.Println("[Error]:", err)
		os.Exit(1)
	}
}
```

Функция Exit находится в стандартном пакете os.

4. *Функции EncryptRSA и DecryptRSA. Используют пакеты* ***‘crypto/rsa’****,* ***‘crypto/rand’*** *и* ***‘crypto/sha256’***

```
func EncryptRSA(data []byte, pub *rsa.PublicKey) ([]byte, error) {
   return rsa.EncryptOAEP(sha256.New(), rand.Reader, pub, data, nil)
}
func DecryptRSA(data []byte, priv *rsa.PrivateKey) ([]byte, error) {
   return rsa.DecryptOAEP(sha256.New(), rand.Reader, priv, data, nil)
}
```

Шифрование данными функциями возможно лишь 190 байт информации, иными словами пригодно для обмена сеансовыми ключами. Первый аргумент функций – это данные которые нужно либо зашифровать, либо расшифровать. Второй аргумент – это либо публичный ключ (при шифровании), либо приватный (при расшифровании).

5. *Цифровые подписи. Используются пакеты* ***‘crypto’****,* ***‘crypto/rsa’*** *и* ***‘crypto/rand’****.*

```
func SignRSA(priv *rsa.PrivateKey, data []byte) ([]byte, error) {
   return rsa.SignPSS(rand.Reader, priv, crypto.SHA256,
HashSum(data), nil)
}
func VerifyRSA(pub *rsa.PublicKey, data, sign []byte) error {
   return rsa.VerifyPSS(pub, crypto.SHA256, HashSum(data),
sign, nil)
}
```

Подпись производиться должна только на хеше данных. Это убирает две проблемы, где первая – это уязвимость, если ключи используются как для шифрования, так и для подписания, а вторая – длина сообщения.

6. *Помимо шифрования/расшифрования и подписей асимметричными ключами, их также необходимо хранить вне программы, в файлах. Таким образом, мы можем закодировать приватный и публичный ключ для последующего использования.*

*Используются пакеты ‘**crypto/rsa**’, ‘**crypto/x509**’ и ‘**encoding/pem**’.*

Функции EncodePrivate и EncodePublic:

```
func EncodePrivate(priv *rsa.PrivateKey) []byte {
    return pem.EncodeToMemory(
        &pem.Block{
            Type: "RSA PRIVATE KEY",
            Bytes: x509.MarshalPKCS1PrivateKey(priv),
        },
    )
}

func EncodePublic(pub *rsa.PublicKey) []byte {
    return pem.EncodeToMemory(
        &pem.Block{
            Type: "RSA PUBLIC KEY",
            Bytes: x509.MarshalPKCS1PublicKey(pub),
        },
    )
}
```

Функции принимают либо публичный, либо приватный ключ и переводят их в стандарт pem.

Чтобы раскодировать приватный и публичный ключ, можно применить функции DecodePrivate, DecodePublic:

*Они используются пакеты ‘**crypto/rsa**’, ‘**crypto/x509**’ и ‘**encoding/pem**’.*

```
func DecodePrivate(data string) *rsa.PrivateKey {
    block, _ := pem.Decode([]byte(data))

    priv, err := x509.ParsePKCS1PrivateKey(block.Bytes)
    utils.CheckError(err)

    return priv
}

func DecodePublic(data string) *rsa.PublicKey {
    block, _ := pem.Decode([]byte(data))

    pub, err := x509.ParsePKCS1PublicKey(block.Bytes)
    utils.CheckError(err)

    return pub
}
```

***7.** AES-CBC шифрование/расшифрование.*
*Используются пакеты в шифровании: ‘**io**’, ‘**bytes**’, ‘**crypto/aes**’, ‘**crypto/rand**’, ‘**crypto/cipher**’, ‘**encoding/hex**’.*

```
func EncryptAES(data, key []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
```

```
    }

    blockSize := block.BlockSize()
    data = PKCS5Padding(data, blockSize)

    cipherText := make([]byte, blockSize + len(data))

    iv := cipherText[:blockSize]
    if _, err := io.ReadFull(rand.Reader, iv); err != nil {
        panic(err)
    }

    mode := cipher.NewCBCEncrypter(block, iv)
    mode.CryptBlocks(cipherText[blockSize:], data)

    return cipherText, nil
}
```

В данном случае, создаётся новый объект cipher, в который помещается ключ. Далее вычисляется длина ключа. Если она не равняется 128/192/256 битам, то произойдёт ошибка. В данном случае, для ключа динамической длины лучше использовать предварительное хеширование при помощи хеш-функции SHA256 и получить на выходе ключ в 256 бит. Соответственно, если вам необходимо также захешировать ключ для последующего доступа, то необходимо получившийся хеш хешировать ещё раз. Иными словами:

H(K) – новый ключ в 256 бит используемый для шифрования/расшифрования.
H(H(K)) – хеш значение ключа используемое для проверки ключа.

Функция PKCS5Padding дополняет блок до длины кратной 128/192/256 бит и выглядит она следующим образом:

```
func PKCS5Padding(ciphertext []byte, blockSize int) []byte {
    padding := blockSize - len(ciphertext) % blockSize
    padtext := bytes.Repeat([]byte{byte(padding)}, padding)
    return append(ciphertext, padtext...)
}
```

Далее выделяется место в памяти для зашифрованного сообщения и вычисляется вектор инициализации iv, который заполняется псевдослучайными байтами.
И в конце мы будем шифровать данные при помощи режима сцепления блоков (CBC).

Этой функцией уже можно пользоваться, тем не менее, можно создать новую функцию, которая будет использовать вышеприведённую, но на вход будет приниматься строка и возвращаться строка, а не байты. Соответственно понадобится какая-либо кодировка, которая способна из набора байт вернуть набор символов. В данном случае, я воспользоваться hex.

```
func Encrypt(session_key []byte, data string) string {
    result, _ := EncryptAES(
        []byte(data),
```

```
        session_key,
    )
    return hex.EncodeToString(result)
}
```

*Расшифрование происходит аналогичным способом. Используются следующие пакеты: ‘**crypto/aes**’, ‘**crypto/cipher**’, ‘**encoding/hex**’.*

```
func DecryptAES(data, key []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }
    blockSize := block.BlockSize()

    if len(data) < blockSize {
        panic("ciphertext too short")
    }

    iv := data[:blockSize]
    data = data[blockSize:]

    if len(data)%blockSize != 0 {
        panic("ciphertext is not a multiple of the block size")
    }

    mode := cipher.NewCBCDecrypter(block, iv)
    mode.CryptBlocks(data, data)
```

```
    return PKCS5Unpadding(data), nil
}
```

Функция PKCS5Unpadding выглядит следующим образом:

```
func PKCS5Unpadding(origData []byte) []byte {
    length := len(origData)
    unpadding := int(origData[length-1])
    return origData[:(length - unpadding)]
}
```

И чтобы расшифровать строку, то можно создать следующую функцию расшифрования:

```
func Decrypt(session_key []byte, data string) string {
    decoded, _ := hex.DecodeString(data)
    result, _ := DecryptAES(
        decoded,
        session_key,
    )
    return string(result)
}
```

3. Сетевое программирование

Большим преимуществом Go над многими языками является лёгкое написание программ связанных с сетевым программированием. Не составляет проблем поднять HTTP сервер и наклепать по быстрому сайт без всеразличных сторонних приложений подобия Apache, NGINX и прочего. Также не составляет проблем сделать сервер и клиент, передающие и принимающие сообщения через сокеты под средством транспортных протоколов TCP/UDP. И в данном разделе мы будем всем этим заниматься.

Начнём пожалуй с сокетов и работы с ними. Для этого создадим две программы server.go и client.go как простой пример того, что клиент способен взаимодействовать с сервером.

Сервер будет работать на порту 8080. По умолчанию стоит IP 127.0.0.1, таким образом, необязательно указывать IP, если вся работа будет происходить на одном компьютере. Используем протокол TCP.

1. *Простая реализация.*

```
// server.go
package main

import (
	"os"
	"fmt"
	"net"
)
func main() {
	// Прослушиваем порт 8080
	listen, err := net.Listen("tcp", ":8080")

	// Закрываем прослушиванием сервером
	defer listen.Close()

	// Если ошибка, тогда закрыть программу
	if err != nil {
		os.Exit(1)
	}
	fmt.Println("Server is listening…")

	// Бесконечный цикл
	for {
		// Ожидание связи
		conn, err := listen.Accept()

		// Если произошла ошибка с соединением
		// на стороне сервера – выйти из цикла.
```

```
            if err != nil { break }

            // Запустить функцию в качестве горутины
            // для возможности множественного
            // соединения
            go server(conn)
        }
}

func server(conn net.Conn) {
        conn.Write([]byte("hello, world"))
        conn.Close()
}

// client.go
package main

import (
    "os"
    "fmt"
    "net"
)

func main() {
        // Установка соединения с сервером
        conn, err := net.Dial("tcp", ":8080")

        // По завершении закрыть соединение
        defer conn.Close()
```

```
	// Если произошла ошибка,
	// тогда закрыть программу
	if err != nil {
		os.Exit(1)
	}

	// Установка буфера в 256 байт для принятия
	// сообщения от сервера
	var buffer = make([]byte, 256)
	var message string

	// Бесконечный цикл
	for {
		// Чтение принятых данных по 256 байт
		length, err := conn.Read(buffer)

		// Если связь прекращена – остановить цикл
		if err != nil { break }

		// Иначе читать сообщение
		message += string(buffer[:length])
	}

	fmt.Println(message)
}
```

1. Компилируем

```
$ go build server.go
$ go build client.go
```

2. Запускаем
$ ./server
> Server is listening…

$ ./client
> hello, world

Теперь немного усложним программы и сделаем так, чтобы клиент мог отправлять серверу какое-либо сообщение, а тот в свою очередь переводил все символы в заглавный регистр и отправлял обратно клиенту.

2. *Улучшенная реализация*

// **server.go**
```
package main

import (
	"os"
	"fmt"
	"net"
	"strings"
)

const END_DATA = "[END-DATA]"

func main() {
	listen, err := net.Listen("tcp", ":8080")
	defer listen.Close()
```

```
	if err != nil {
		os.Exit(1)
	}
	fmt.Println("Server is listening...")
	for {
		conn, err := listen.Accept()
		if err != nil { continue }
		go server(conn)
	}
}

func server(conn net.Conn) {
	defer conn.Close()

	var buffer = make([]byte, 256)
	var message string

	for {
		length, err := conn.Read(buffer)
		if err != nil {
			return
		}
		message += string(buffer[:length])
		if strings.HasSuffix(message, END_DATA) {
			message = strings.TrimSuffix(message,
END_DATA)
			break
		}
	}
```

```
        conn.Write([]byte(strings.ToUpper(message)))
}

// client.go
package main

import (
        "os"
    "fmt"
    "net"
)

const END_DATA = "[END-DATA]"

func main() {
    conn, err := net.Dial("tcp", ":8080")
    defer conn.Close()
    if err != nil {
        os.Exit(1)
    }

    conn.Write([]byte("hello, world" + END_DATA))

    var buffer = make([]byte, 256)
    var message string

    for {
        length, err := conn.Read(buffer)
        if err != nil { break }
        message += string(buffer[:length])
```

```
    }

    fmt.Println(message)
}
```

Данная связь немного посложнее, так как здесь идут некоторые дополнительные детали проектирования как со стороны клиента, так и со стороны сервера.

Первое новшество, какое присутствует в этих программах – это константа END_DATA. В прошлом примере, такая константа не нужна была, так как лишь клиент ожидал ответа от сервера. Сервер же в свою очередь сам генерировал текст и отправлял клиенту, далее он прерывал соединение и клиент, понимая что соединение прервано, удостоверивался в получении всей информации от сервера и выводил её на экран.

В данном же случае идёт двусторонняя связь, при которой сервер должен читать посылаемые данные клиентоми и при этом не должен прекращать соединение. Соответственно ему необходимо знать, когда закончатся посылаемые данные. В данном случае константа END_DATA как раз и является окончанием сообщения. Понимая это, сервер начинает уже обрабатывать данные и посылать их обратно клиенту и после отправки всех данных закрывает соединение. Клиент понимая, что соединение закрыто, читает все данные отправленные сервером и выводит на экран.

Если клиент отправит сообщение серверу без константы END_DATA, тогда сервер будет пытаться найти END_DATA до тех пор, пока клиент не пошлёт ему эту константу, либо до тех пор, пока связь с клиентом не будет разорвана. Данный сервер можно ещё улучшить тем, что связь будет прерываться через определённое количество времени с начала соединения.

Функция HasSuffix проверяет наличие константы в конце посланного сообщения.
Функция TrimSuffix удаляет коснтанту из сообщения.
Функция ToUpper переводит все символы сообщения в верхний регистр.

На основе таких программ уже можно выстраивать определённую структуру поведения как сервера, так и клиента.

3. *Реализация чата на основе архитектуры клиент - сервер.*

Константы завершённости можно и не использовать, если принятие данных возможно без полноценной обработки, как пример, в чате, где данные просто пересылаются сервером.

```
// server.go
package main

import (
    "os"
    "log"
    "net"
    "time"
)

const (
    PROTOCOL = "tcp"
    PORT = ":8080"
    BUFF = 256
    QUAN = 2
)

var connections = make(map[net.Conn]bool)

func main() {
    listen, err := net.Listen(PROTOCOL, PORT)
    if err != nil {
        os.Exit(1)
    }
    defer listen.Close()

    log.Println("[Server is listening]")
    for {
```

```
        // Если количество соединений меньше, чем
        // константное число максимума, тогда разрешить
        // связь
        if len(connections) < QUAN {
            conn, err := listen.Accept()
            if err != nil { break }

            // Отправить клиенту подтверждение на
            // разрешение соединения
            tryConnection(conn)
            go connection(conn)
        }
        time.Sleep(500 * time.Millisecond)
    }
}

func tryConnection(conn net.Conn) {
    conn.Write([]byte{1})
}

func connection(conn net.Conn) {
    // Закрыть соединение по окончанию работы функции
    defer conn.Close()

    // Добавление соединения в словарь
    connections[conn] = true
    var buffer []byte = make([]byte, BUFF)
    for {
        // Чтение сообщения от клиента
        length, err := conn.Read(buffer)
```

```
        if err != nil { break }
        log.Print(string(buffer[:length]))

        // Перебор всех клиентов и отправление всем
        // одного сообщения
        for user := range connections {
            if user != conn {
                user.Write(buffer[:length])
            }
        }
    }

    // Удаление связи из словаря
    delete(connections, conn)
}

// client.go
package main

import (
    "os"
    "fmt"
    "net"
    "time"
    "bufio"
    "strings"
)

const (
    PROTOCOL = "tcp"
```

```
    PORT = ":8080"
    BUFF = 256
)

var username string

func main() {
    conn, err := net.Dial(PROTOCOL, PORT)
    if err != nil {
        os.Exit(1)
    }
    defer conn.Close()

    // Попытаться соединиться с сервером
    tryConnection(conn)
    username = inputString("Nickname: ")

    // Ввод данных
    go client(conn)

    // Чтение данных
    readMessages(conn)
}

func tryConnection(conn net.Conn) {
    var check = make([]byte, 1)

    // Установить таймера на одну секунду и запустить
    var timer = time.NewTimer(time.Second)
```

```
    // Если время на таймере вышло, тогда выдать
    // ошибку о невозможности подключиться
    // к серверу
    go func() {
        <- timer.C
        fmt.Println("Connection failure")
        os.Exit(1)
    }()

    // Если пришёл ответ от сервера, тогда остановить
    // таймер
    conn.Read(check)
    timer.Stop()
    fmt.Println("Connection success")
}

// Ввод сообщения и отправка
func client(conn net.Conn) {
    for {
        var message = inputString("")
        if len(message) != 0 {
            conn.Write([]byte(
                fmt.Sprintf("[%s]: %s\n", username, message),
            ))
        }
    }
}
```

```
// Принятие и чтение данных
func readMessages(conn net.Conn) {
    var buffer = make([]byte, BUFF)
    for {
        length, _ := conn.Read(buffer)
        if length != 0 {
            fmt.Print(string(buffer[:length]))
        }
    }
}

// Ввод текста
func inputString(text string) string {
    fmt.Print(text)
    message, _ := bufio.NewReader(os.Stdin).ReadString('\
n')
    return strings.Replace(message, "\n", "", -1)
}
```

Новшества:
1. Вынос констант за пределы функции main.
2. Вынос переменных из локальных в глобальные, для удобства работы.
3. Ограниченное количество соединений сервера.
4. Подтверждение соединения сервером.
5. Отправка сообщения всем клиентам.
6. Чтение и отправка сообщений без прерывания соединения и без внедрения константы окончания данных.

Также стоит сказать, хоть мы и работаем с глобальной переменной при помощи горутин (на сервере), нам необязательно в данном случае использовать функции Lock и Unlock из пакета sync. Так как работы горутин с данными в переменной connections не пересекается.

Минус у данной схемы отправки сообщений присутствует (где нет константы окончания). Если вы захотите дополнить ваш функционал, то необходимо будет уместить его в первые 256 байт информации (от ёмкости буффера). Также, будет существовать проблема со стороны криптографии, так как будут пригодны лишь поточные шифры, либо же блочные шифры с режимом шифрования ECB, что как итог уменьшает защищённость соединения.

4. *Реализация чата на основе P2P сети.*

Вышеописанный способ соединения достаточно проблематично реализовывать в P2P.

Это обусловленно некоторыми нюансами. Допустим, в архитектуре клиент-сервер, сразу предполагается, что сервер обладает большим объёмом ресурсов для обработки данных и принятия/сохранения соединений. Клиенту же в данной архитектуре нужно лишь поддерживать связь с одним сервером.

Проблемы реализации P2P по аналогии с клиент-серверной архитектурой следующие:

1. Если мы предпримем двустороннее соединение в P2P, то клиент по-факту будет являться сервером, хранящим все соединения и ожидающим от всех сообщений. К тому же, пользователь P2P не теряет свойств клиента и потому, со стороны клиента также необходимо организовать связь с другими “серверами” в сети.

2. Также стоит сказать, что сеть P2P способна очень быстро разрастаться при условии того, что используется распределённая хеш-таблица (DHT). Как итог, обычные клиенты, у которых нет мощностей серверов, будут быстро отваливаться по мере увеличения объёма сети.

3. При смещении сервера и клиента в одно целое, формируется чёткая граница между ними, где сервер способен только принимать данные, а клиент только отправлять. В то время, как в клиент-серверной архитектуре, как клиент, так и сервер могут принимать данные и отправлять их. Таким образом, если мы будем использовать клиент-серверную архитектуру в P2P, то будет предполагаться наличие неиспользуемого функционала, а также более сложная расширяемость приложения со стороны программирования.

Как итог, куда лучше для данного вида сети использовать самый первый способ, где передача сообщений происходила без двустороннего соединения.

У такой архитектуры безусловно существует минус, а именно при отправке сообщения необходимо будет каждый раз подключаться к “серверу”, отправлять данные и закрывать соединение. Такая же проблема будет со стороны серверной части, где мы будем принимать связь и сообщение, а после принятия завершать соединение. Тем самым теряется скорость связи, но взамен этому отбрасывается три вышеперечисленных минуса.

// **clientP2P.go**

```
package main

import (
    "os"
    "fmt"
    "net"
    "bufio"
    "strings"
    "encoding/json"
)

const (
    PROTOCOL = "tcp"
    BUFF = 256
)
```

```
type packageTCP struct {
    From string
    Body string
}

var (
    connections = make(map[string]bool)
    address string
)

func main() {
    initArgs(os.Args)
    go client()
    server()
}

func initArgs(args []string) {
    var flag_address bool
    for _, value := range args {
        switch value {
            case "-a", "--address":
                flag_address = true
                continue
        }
        switch {
            case flag_address:
                address = value
                flag_address = false
        }
    }
```

```
    if address == "" {
        printError("address undefined")
    }
}

func client() {
    for {
        var message = inputString()
        var splited = strings.Split(message, " ")
        switch splited[0] {
            case ":exit": os.Exit(0)
            case ":network": network()
            case ":connect":
                if len(splited) > 1 {
                    connectTo(splited[1:])
                }
            case ":disconnect":
                if len(splited) > 1 {
                    disconnectFrom(splited[1:])
                }
            default:
                for addr := range connections {
                    sendPacket(addr, message)
                }
        }
    }
}
```

```
func server() {
    listen, err := net.Listen(PROTOCOL, address)
    if err != nil {
        printError("can't run listener")
    }
    defer listen.Close()
    fmt.Println("[ClientP2P is run]")
    for {
        conn, err := listen.Accept()
        if err != nil {
            break
        }
        go handleConnect(conn)
    }
}

func handleConnect(conn net.Conn) {
    defer conn.Close()
    var (
        buffer = make([]byte, BUFF)
        message string
        pack packageTCP
    )
    for {
        length, err := conn.Read(buffer)
        if err != nil { break }
        message += string(buffer[:length])
    }
    json.Unmarshal([]byte(message), &pack)
    connectTo([]string{pack.From})
```

```
    fmt.Printf("[%s]: %s\n", pack.From, pack.Body)
}

func sendPacket(to, message string) {
    conn, err := net.Dial(PROTOCOL, to)
    if err != nil {
        disconnectFrom([]string{to})
        return
    }
    var pack = packageTCP {
        From: address,
        Body: message,
    }

    data, err := json.Marshal(pack)
    if err != nil {
        printError("can't convert pack to json")
    }

    conn.Write(data)
    conn.Close()
}

func network() {
    for addr := range connections {
        fmt.Println("|", addr)
    }
}
```

```
func connectTo(conn []string) {
    for _, value := range conn {
        connections[value] = true
    }
}

func disconnectFrom(conn []string) {
    for _, value := range conn {
        delete(connections, value)
    }
}

func inputString() string {
    message, _ := bufio.NewReader(os.Stdin).ReadString('\
n')
    return strings.Replace(message, "\n", "", -1)
}

func printError(err string) {
    fmt.Println("[Error]:", err)
}
```

1. Компиляция программы:
$ go build clientP2P.go

2. Запуск клиента:
[Первое окно терминала]
$ ./clientP2P -a :8080

[Второе окно терминала]
$ ./clientP2P -a :9090
> :connect :8080
> hello! // Ввод

[Первое окно терминала]
> hello! // Получение
> hi! // Ввод

[Второе окно терминала]
> hi! // Получение

5. *HTTP-сервер.*

Возможности HTTP-сервера носят два характера.
Первый – это возможность создания GUI, где приоритет – это отображение информации в уже готовой программе.
Второй – это возможность создания сайта, где приоритет – это создание и поднятие сервера.

Если речь идёт про P2P-соединение, то выбирают конечно же первый вариант. Тем не менее, второй вариант не исключается, допустим мы можем реализовать сеть P2P поверх сети Tor. Тем самым исключаются некоторые проблемы связанные с шифрованием, так как сами сервера Tor’а будут служить нам центром распределения ключей и сами сервисы Tor’а будут генерировать нам как публичный (имя сайта), так и приватный ключи. Таким образом, мы

сможем убить двух зайцев создав как сайт, на который будут переходить пользователи, так и программу, при помощи которой мы сможем связываться с другими.
Но так или иначе, данный пример скорее исключение из правил, нежели действенный вариант. Хоть мы и создаём сеть P2P поверх сети Tor, она всёравно завязана на серверах (хоть они также и находятся в P2P соединении).

Один из самых простых вариантов HTTP-сервера выглядит следующим образом:

```
package main

import (
    "os"
    "fmt"
    "net/http"
)

const PORT = ":7545"

func main() {
    http.HandleFunc("/", indexPage)
    checkError(http.ListenAndServe(PORT, nil))
}

func indexPage(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "hello, world")
}
```

```
func checkError(err error) {
    if err != nil {
        printError(err.Error())
    }
}

func printError(err string) {
    fmt.Println("[Error]:", err)
    os.Exit(1)
}
```

Теперь если вы скомпилируете, запуститите программу и перейдёте в браузер по адресу “localhost:7545”, то увидите сообщение “hello, world”.

Попробуем усложнить задачу, сделав маршрутизацию новых страниц, логирование и добавление view + статических файлов.

Для начала создадим директорию views для файлов html. Далее создадим четыре файла: base.html, index.html, news.html, about.html и page404.html.

```
// base.html
<!DOCTYPE html>
<html>
<head>
        <title>
                {{ block "title" . }}
                {{ end }}
```

```
        </title>
        <meta charset="utf-8">
        <link rel="stylesheet" type="text/css" href="/static/css/
style.css">
</head>
<body>
        <header>
                <div align="center">
                        <h2>Site for 5 minutes</h2>
                </div>
                <br>
                <nav>
                        |
                        <a href="/">Home</a> |
                        <a href="/news">News</a> |
                        <a href="/about">About</a>
                        |
                </nav>
        </header>
        <main>
                <hr>
                {{ block "body" . }}
                {{ end }}
                <hr>
        </main>
        <footer>
                <p>[example]</p>
        </footer>
</body>
</html>
```

// **index.html**

```
{{ define "title" }}
    Home
{{ end }}

{{ define "body" }}
    <p>hello, world</p>
{{ end }}
```

// **news.html**

```
{{ define "title" }}
    News
{{ end }}

{{ define "body" }}
    <form method="POST" action="/news">
        <input type="text" name="title"
placeholder="Title..."> <br>
        <input type="text" name="body"
placeholder="Body..."> <br>
        <input type="submit" name="add_news"
value="Создать">
    </form>
    {{ range .NewsData }}
        <hr>
        <div>
            <h3>Title: {{ .Title }}</h3>
            <p>Body: {{ .Body }}</p>
        </div>
    {{ end }}
```

```
{{ end }}

// about.html
{{ define "title" }}
    About
{{ end }}

{{ define "body" }}
    <p>Just example for about</p>
{{ end }}

// page404.html
{{ define "title" }}
    Page 404
{{ end }}

{{ define "body" }}
    <p>Nothing</p>
{{ end }}
```

А также создадим директорию static. В ней создадим директорию css. И в директории css создадим файл style.css.

```
// style.css
footer > * {
    color: blue;
}
```

И наконец создадим основной файл. Назовём его main.go.

```
// main.go
package main

import (
    "os"
    "fmt"
    "log"
    "net/http"
    "html/template"
)

const (
    PATH_VIEWS = "views/"
    PATH_STATIC = "static/"
    PORT = ":7545"
)

// Пакет для отображения новостей
type newsPacket struct {
    Title string
    Body string
}

// Стандартные данные для раздела новостей
var newsData = []newsPacket {
    { Title: "Title1", Body: "Body1" },
    { Title: "Title2", Body: "Body2" },
```

```
    { Title: "Title3", Body: "Body3" },
}

func main() {
    routerHTTP()
    checkError(http.ListenAndServe(PORT,
logger(http.DefaultServeMux)))
}

// Перехват функции и перед её обработкой
// выводить информацию о методе и URL адресе
func logger(handler http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        log.Printf("[%s] %s\n", r.Method, r.URL)
        handler.ServeHTTP(w, r)
    })
}

func routerHTTP() {
    // Просмотр всей статической директории, где
    // хранятся css и img файлы.
    http.Handle("/static/", http.StripPrefix(
        "/static/",
        handleFileServer(http.Dir(PATH_STATIC))),
    )

    http.HandleFunc("/", indexPage)
    http.HandleFunc("/news", newsPage)
    http.HandleFunc("/about", aboutPage)
```

```
}

func indexPage(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        // Если адрес не равен корневому пути
        // тогда сделать редирект на страницу с 404
        redirectTo("page404", w, r)
        return
    }
    tmpl, err := template.ParseFiles(PATH_VIEWS +
"base.html", PATH_VIEWS + "index.html")
    checkError(err)
    tmpl.Execute(w, nil)
}

func newsPage(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {
        // Добавление новости
        r.ParseForm()
        if _, ok := r.Form["add_news"]; ok {
            newsData = append(newsData, newsPacket{
                Title: r.FormValue("title"),
                Body: r.FormValue("body"),
            })
        }
    }
```

```
    // Установка новостей для отображения
    var data = struct {
        NewsData []newsPacket
    }{
        NewsData: newsData,
    }

    tmpl, err := template.ParseFiles(PATH_VIEWS +
"base.html", PATH_VIEWS + "news.html")
    checkError(err)
    tmpl.Execute(w, data)
}

func aboutPage(w http.ResponseWriter, r *http.Request) {
    tmpl, err := template.ParseFiles(PATH_VIEWS +
"base.html", PATH_VIEWS + "about.html")
    checkError(err)
    tmpl.Execute(w, nil)
}

func redirectTo(to string, w http.ResponseWriter, r
*http.Request) {
    switch to {
        case "page404": page404(w, r)
    }
}
```

```
// Проверка наличия директории static
func handleFileServer(fs http.FileSystem) http.Handler {
    return http.HandlerFunc( func(w http.ResponseWriter, r
*http.Request) {
        if _, err := fs.Open(r.URL.Path); os.IsNotExist(err) {
            redirectTo("404", w, r)
            return
        }
        http.FileServer(fs).ServeHTTP(w, r)
    })
}

func page404(w http.ResponseWriter, r *http.Request) {
    tmpl, err := template.ParseFiles(PATH_VIEWS +
"base.html", PATH_VIEWS + "page404.html")
    checkError(err)
    tmpl.Execute(w, nil)
}

func checkError(err error) {
    if err != nil {
        printError(err.Error())
    }
}

func printError(err string) {
    fmt.Println("[Error]:", err)
    os.Exit(1)
}
```

В идеале, структура newsPacket должна находиться в пакете models, а сами данные newsPacket должны находиться в базе данных, константы должны находиться в пакете settings, функция подобия printError, checkError и прочие должны находиться в пакете utils, indexPage, newsPage, aboutPage, page404 должны находиться в пакете controllers и так далее. Но из-за необходимости в краткости сделано всё в одном файле.

4. Дополнительная литература

“Искусство программирования. Том 1. Основные алгоритмы.” (Д. Кнут).
“Структура и интерпретация компьютерных программ” (Х. Абельсон, Дж. Сассман).

5. Практические задания

Задания расположены по мере возрастания сложности в каждом из данных разделов:

I. Криптография с открытым ключом и криптографические протоколы:

1. Реализовать головоломки Меркла (1 глава, 7 раздел, 1 пункт). Количество сообщений = 1000, ключи (для шифрования сообщений) случайные и размером в 24 бита (3 байта).

2. Реализовать протокол Диффи-Хеллмана (1 глава, 6 раздел, 1 пункт), при помощи функций (3 глава, 1 раздел).

3. Реализовать алгоритм RSA (1 глава, 6 раздел, 3 пункт), при помощи функций (3 глава, 1 раздел).

4. Реализовать алгоритм Эль-Гамаля (1 глава, 6 раздел, 2 пункт), при помощи функций (3 глава, 1 раздел).

5. Реализовать один из протоколов (1 глава, 7 раздел, 4 пункт), при помощи функций (3 глава, 2 раздел).

6. Реализовать алгоритм Меркла-Хеллмана (1 глава, 6 раздел, 4 пункт), при помощи функций (3 глава, 1 раздел).

II. *P2P сеть:*

1. В файле clientP2P.go (3 глава, 3 раздел, 4 пункт) добавить возможность просматривать наличие файлов у определённого пользователя, а также скачивать их. Файлы должны распологаться только в директории Archive.

2. Связать файл clientP2P.go (3 глава, 3 раздел, 4 пункт) с HTTP-сервером для отображения данных: соединений и сообщений.

3. Реализовать end-to-end шифрование в файле clientP2P.go (3 глава, 3 раздел, 4 пункт). End-to-end шифрование предполагает шифрование на стороне отправляющего и расшифрование на стороне принимающего. Функции шифрования: (3 глава, 2 раздел).