Прикладная Криптография: Симметричные криптосистемы IPsec, TLS (SSL)

Макаров Артём

МИФИ 2018

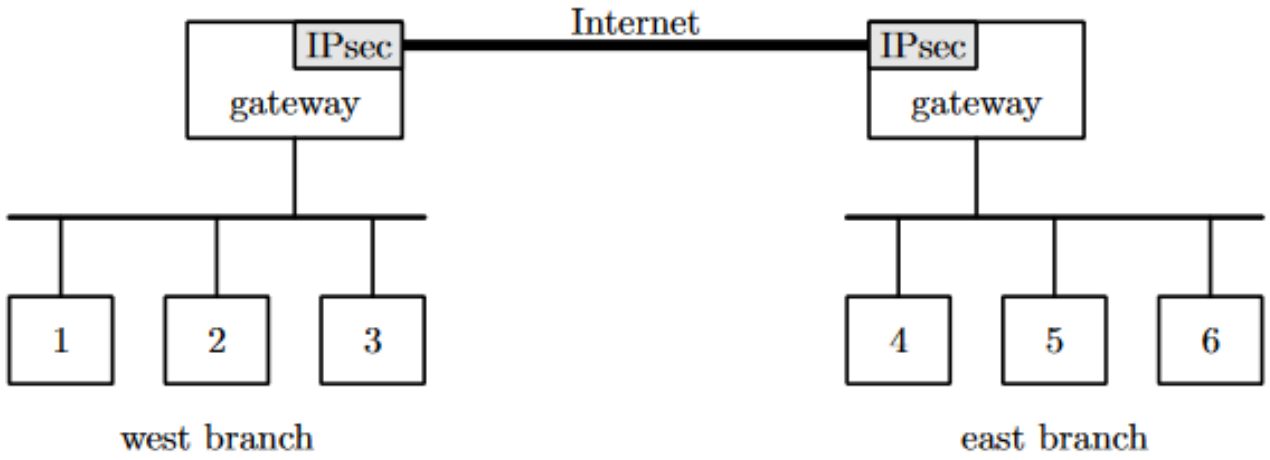
Построение защищенных каналов связи

Одной из задач криптографии является построение защищенных каналов связи, обеспечивающий аутентичность и конфиденциальность передаваемой информации.

Можно выделить 2 части данных протоколов – симметричную, обеспечивающую целостность и конфиденциальность самой передаваемой информации, и ассиметричную, обеспечивающую аутентификацию участников и позволяющую согласовать общий симметричный секрет (сессионный мастер ключ).

IPsec

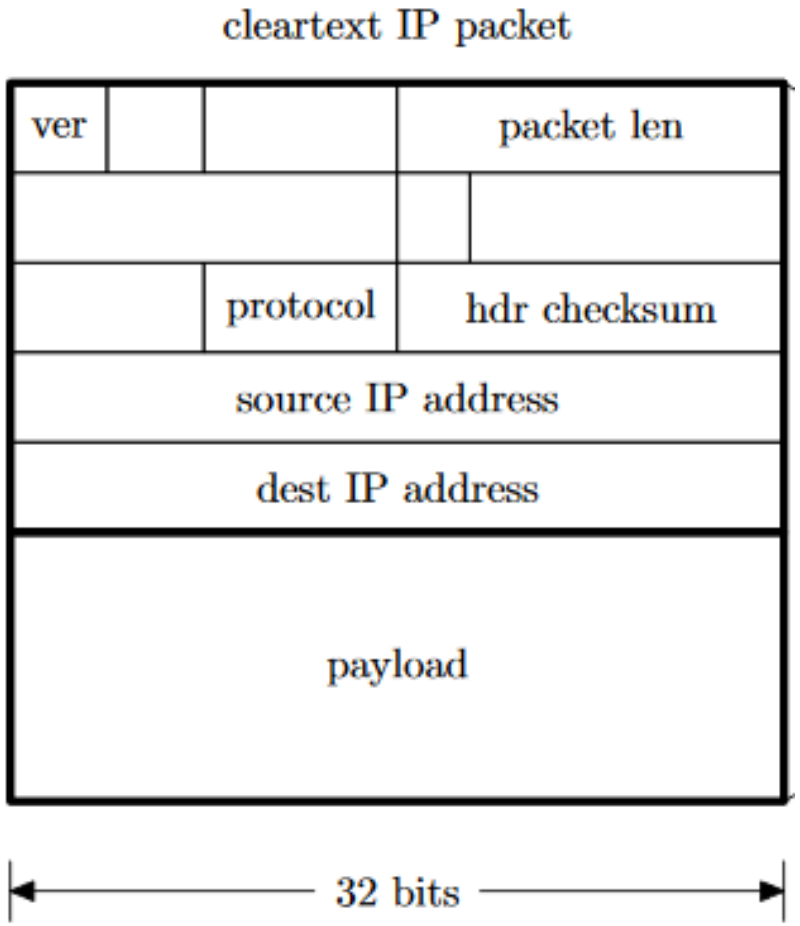
- Обеспечивает целостность и конфиденциальность IP пакетов
- На самом деле – семейство протоколов. Рассмотрим протокол ESP (encapsulated security payload) в режиме тунелирования.
- Используется для построения VPN



IP

Рассмотрим IP пакет для IPv4.

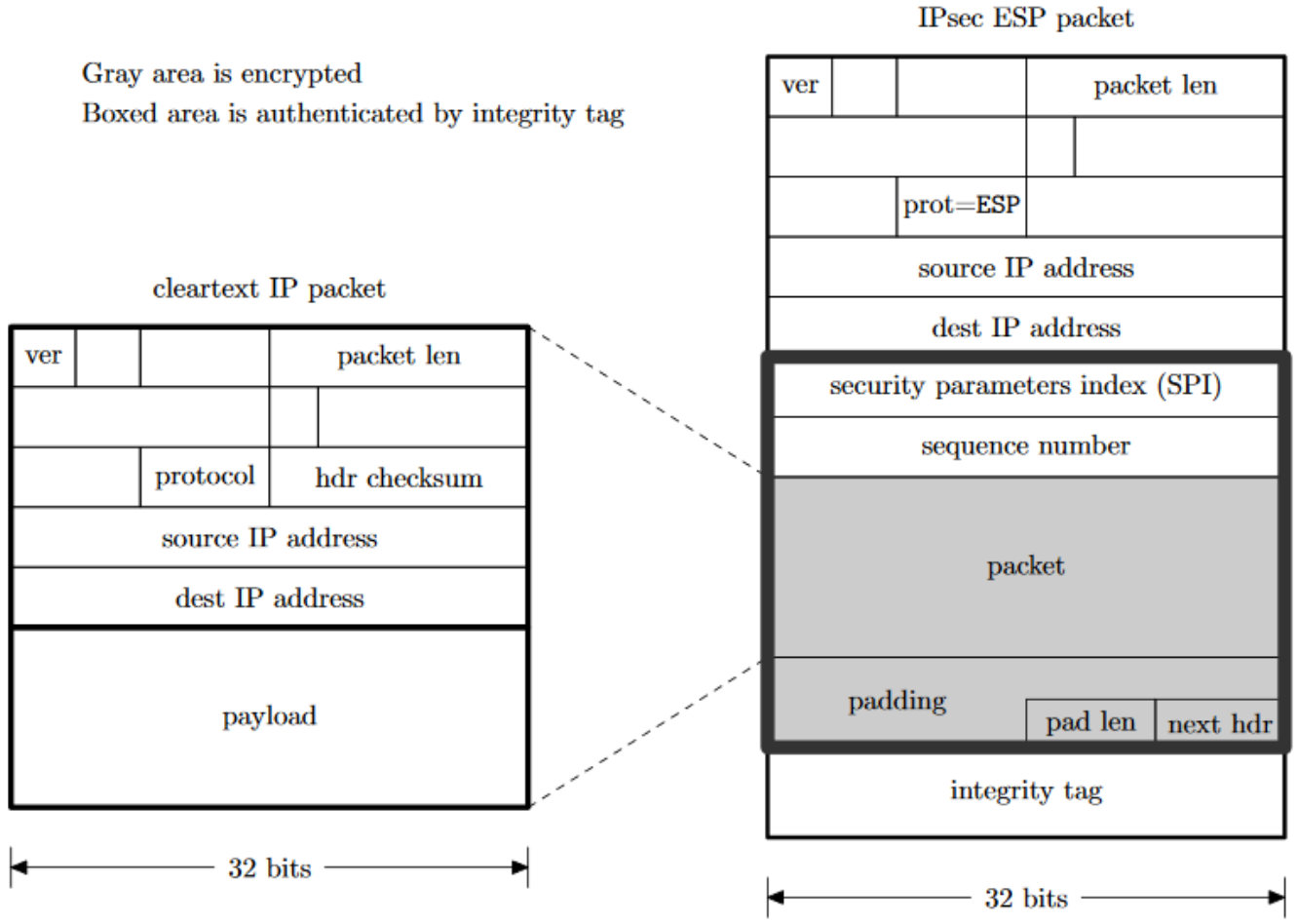
- ver – версия, равна 4 для IPv4 (1 байт)
- packet len – длина **всего** пакета (2 байта)
- prot – описание протокола верхнего уровня (TCP=6)
- hdr checkum – контрольная сумма
- sourse, dest – ip адрес получателя и отправителя пакета
- payload – данные для передачи



Инкапсуляция IPsec

На конечных точках имеет SAD (security association database), записями в которой называются SA (security association), индексируемые 32 битным числом SPI (security parameter index).

SA содержит набор параметров, включающих идентификаторы криптографических алгоритмов, секретные ключи, SPI, адреса получателей и отправителей, параметры обмена ключами

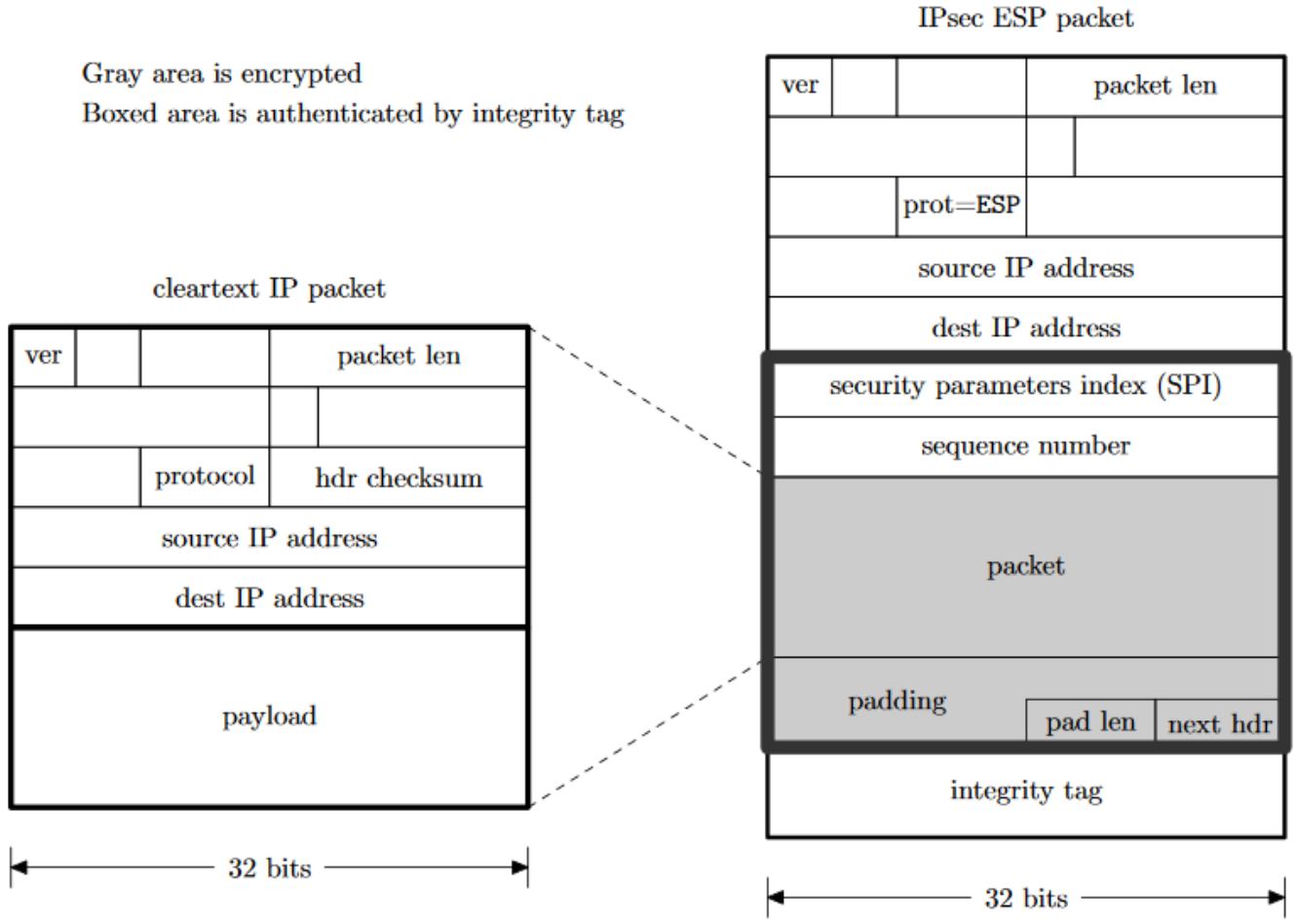


Инкапсуляция IPsec

Для отправки пакета отправитель ищет адрес получателя в SAD, получает параметры соединения и устанавливает защищенный канал, используя данные параметры.

Получатель, при получении пакета:

- Проверяет наличие SA в своей базе по (адрес отправителя, адрес получателя, SPI)
- Если не найдена – проверяет наличие на основе (SPI, адрес получателя)
- Если не найдена – ищет только по SPI
- Если не найдена – отбросить пакет
- Если найдена – расшифровать пакет с использованием ключа, записанного в SA

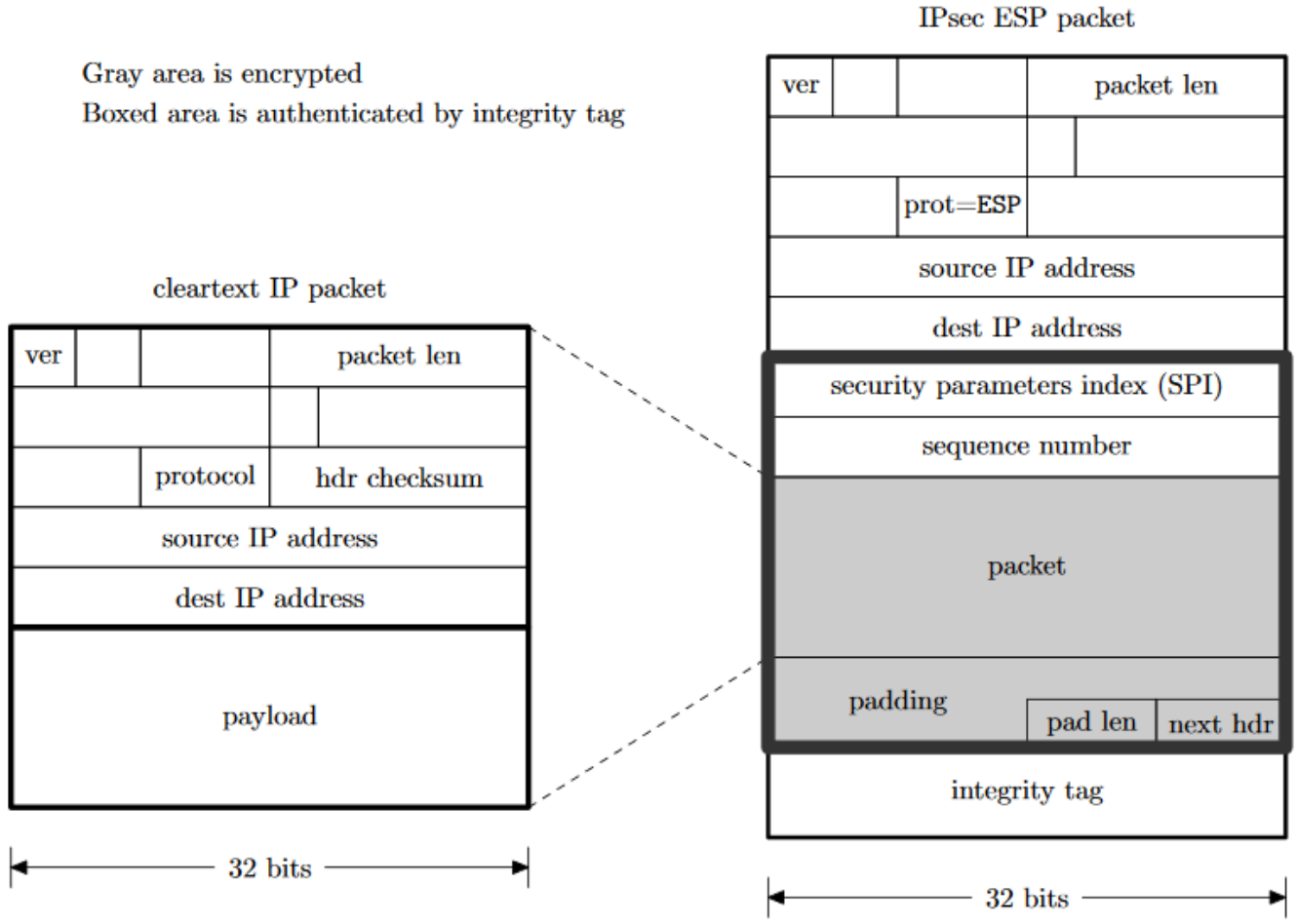


Инкапсуляция IPsec

При двухстороннем соединении при шифровании используются два канала – от отправителя к получателю и от получателя к отправителю. Для них используются различные SA с различными ключами.

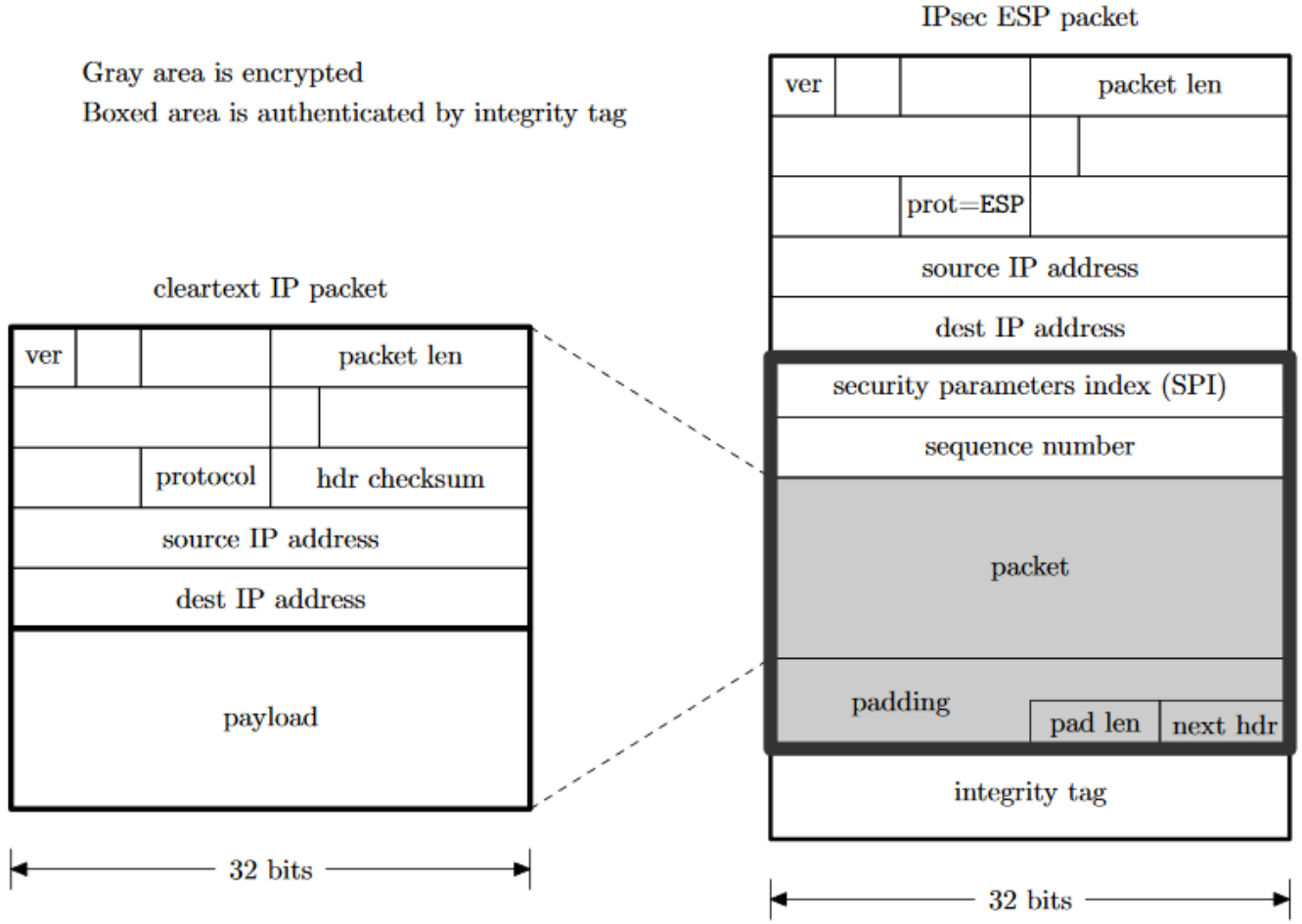
Т.е. в общем случае для каждого соединения в SAD хранятся 2 записи.

Если для соединения хранится только одна запись – соединение одностороннее.



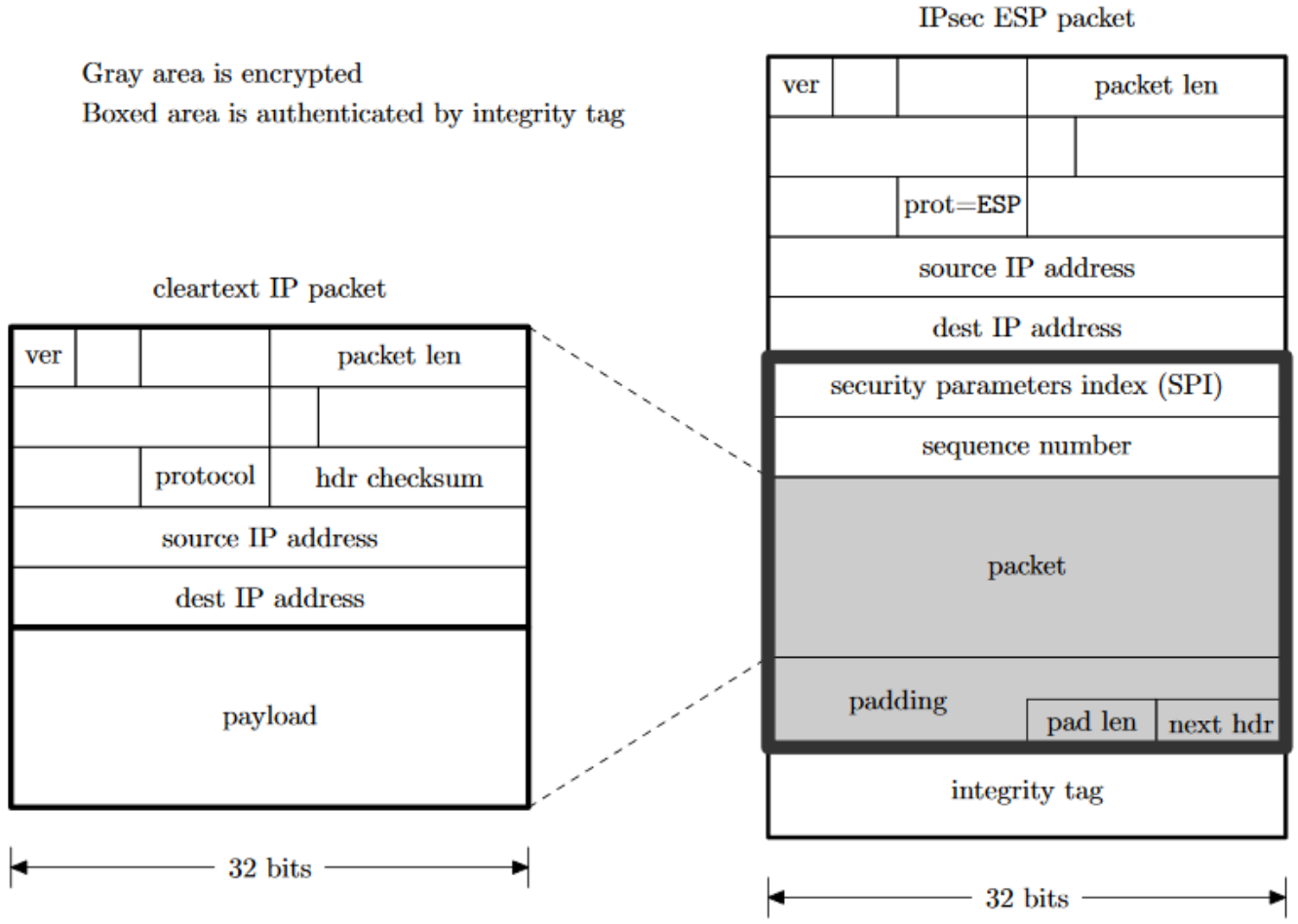
Инкапсуляция IPsec

sequence number – номер пакета, используется для обнаружения и отбрасывания повторяющихся пакетов. 64 бита, но записывается в пакета только наименее значимые 32 бита. При вычислении MAC используются все 64 бита. Инициализируется нулём при установлении соединения, увеличивается на 1 с каждым пакетом.



Инкапсуляция IPsec

- padding – дополнение до длины блока алгоритма шифрования и результирующего шифр текста до длины 4 байта. От 0 до 255 байт.
- pad len – длина дополнения
- next hdr – тип данных (для данного примера IPv4=4)



Шифрование IPsec

MAC-then-Encrypt

- Данные дополняются дополнением до необходимой длины. Заполняется поле next header
- Зашифровываются ключом для данной SA. Если шифрование обозначено как NULL, оно не производится (тогда IPsec обеспечивает только целостность).
- Вычисляется MAC на следующих данных:

  SPI || sequence number (64 бита) || ciphertext
- Инкапсуляция пакета в IP пакет

Прочие хитрости

- TFC дополнение (traffic flow confidentiality) – дополнение, для скрытия размера открытого текста, используется до дополнения до размера блочного шифра, произвольной длины
- Dummy blocks – блоки, не несущие полезной нагрузки, и отбрасываемые получателем при расшифровке.

- Возможно только шифрование, без вычисление MAC
  - Опасно, даже если предположить, что протоколы верхнего уровня обеспечивают целостность (получаем mac-then-encrypt)
  - Безопасно, при использовании аутентичного шифрования

The Cryptographic Doom Principle

- When it comes to designing secure protocols, I have a principle that goes like this: if you have to perform any cryptographic operation before verifying the MAC on a message you've received, it will somehow inevitably lead to doom.

SSH

SSH (secure shell) – утилитка для удалённой консоли. Разработана как защищенная альтернатива telnet. Использует MAC-and-Encrypt

1995, SSHv1. «Что может пойти не так?»

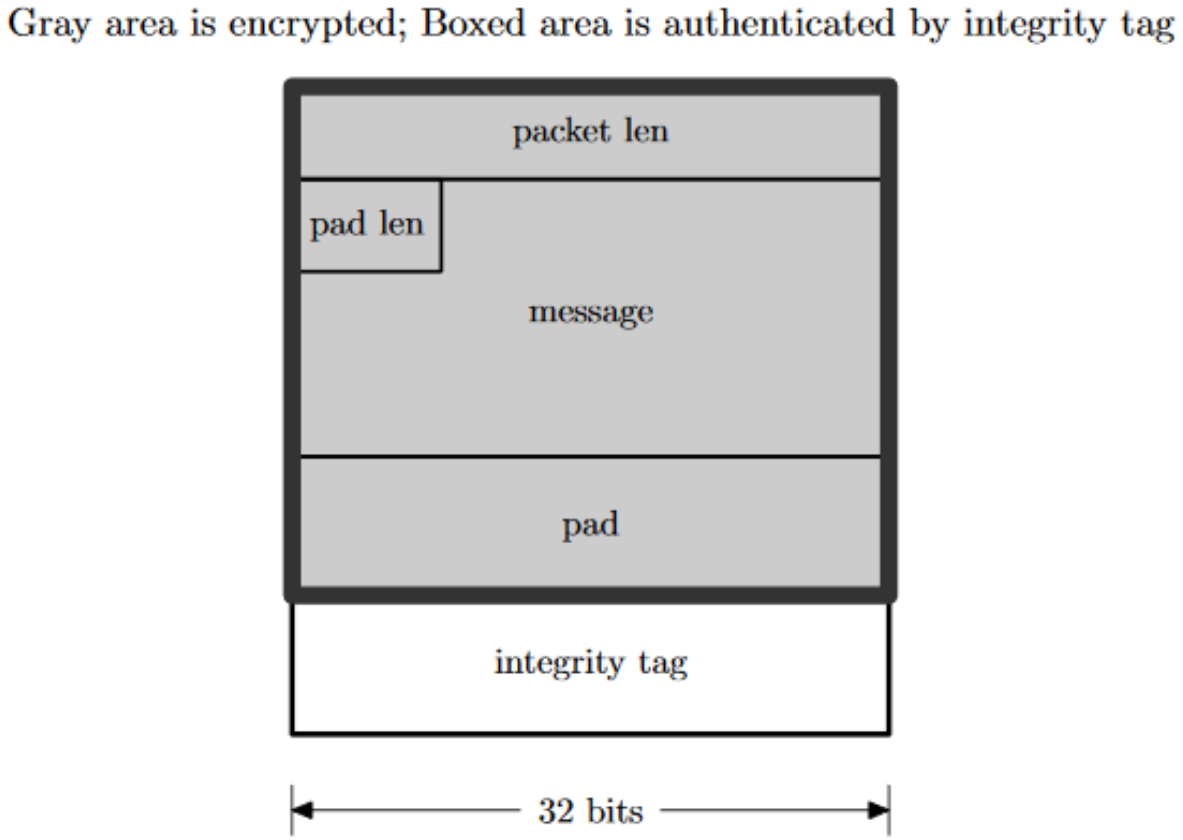
- Обеспечивает целостность данных при передаче используя CRC (что не только позволяет подделать “MAC”, но и узнать часть данных об открытом тексте)
- Использует шифрование CBC с нулевым инициализирующем вектором
- Использует одинаковый ключ для обоих направлений передачи данных
- Использует неатомарное шифрование – расшифрованные происходит поточно, до проверки целостности данных (подробнее – далее).

SSHv2

1996, SSHv2. Исправили большинство проблем.

Теперь использует 2 различных ключа для двух различных направлений передачи данных.

Использует CPA стойкое шифрование и стойкий MAC.



SSHv2

Шифрование:

- Открытый текст дополняется случайными байтами для выравнивание до длины блочного шифра, от 4 до 255 байт

$$plaintext = ||packet\text{-}len||pad\text{-}length||message||pad$$

- Зашифрованные с использованием AES в рандомизированном CBC режиме и использованием симметричного ключа для данного направления (но использует предсказуемый IV для последующих блоков, используя последний блок шифртекста, было исправлено но не сразу)
- Вычисляет MAC для sequence number и plaintext. Множество алгоритмов, включа HMAC-SHA1-160

SSHv2

Расшифрование:

- Расшифрование поля packet length используя ключ, для данного направления.
- Считать packet length + (длина MAC) байт из канала связи
- Расшифровать оставшийся шифртекст
- Проверить MAC

Проблемы и особенности SSHv2

- Некоторые сочетания алгоритмов не являются стойкими
- Используется шифрование длины пакета
  - Используется сокрытие длины пакета
  - Используется для неатомарного расшифрования
  - Исправляется костылями в хороших реализациях (Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm)
- Шифрование «побуквенное», т.е. частота пакетов соответствует частоте нажатия клавиш
  - Частотное восстановление открытого текста
  - Используя «dummy blocks» для защиты

Основной проблемой является использование части открытого текста (длины пакета) до проверки её целостности, что ведёт к атаке.

Атака на неатомарное шифрование

- Пусть противник имеет некоторый 16 байтный шифртекст c.
- Противник отправляет шифртекст внутри ssh пакета на сервер.
- Сервер расшифровывает первые 4 байта и интерпретирует их как количество пакетов, которые необходимо получить
- Противник отправляет побайтно случайные биты серверу, считая их количество
- Сервер, считав необходимое число байт + число байт для MAC, проверяет MAC (который очевидно не сходится) и возвращает ошибку
- Противник зная количество отправленных байтов восстанавливает первые 4 байта шифртекста
- Если шифруется каждое нажатие, то фактически можно читать весь трафик

SSL 3.0

SSL 3.0 – протокол для установления защищенного канала

- Использует Encrypt-then-MAC
- Возможно использование Рандомизированный CBC (CPA стойкий) и стойкий MAC
- Использует дополнение. (CBC с дополнением в схеме Encrypt-then-MAC – не стойкая)
- Сломан, возможна атака на расшифрование.
- SSL 3.0 и TLS 1.0 используется предсказуемый IV для последующих блоков, на основе последнего блока шифртекста

SSL 3.0

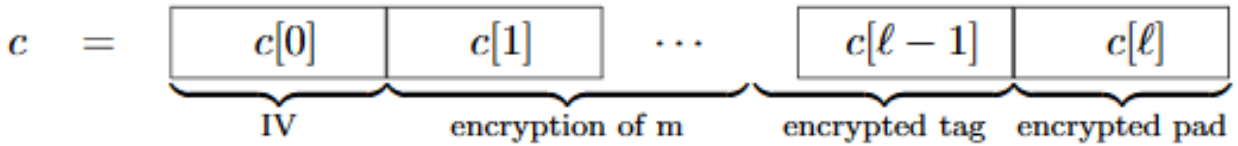
Пусть используется AES в CBC режиме.

Шифрование:

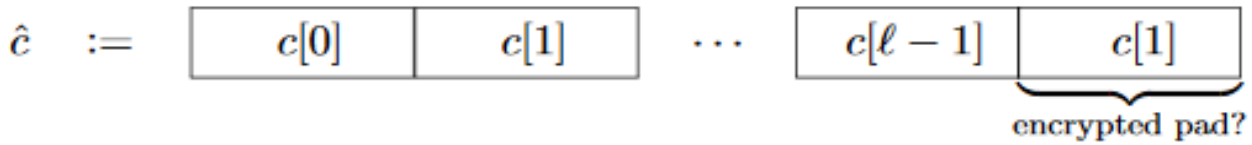
- Вычисляется MAC для сообщения.
- Дополнение. Если требуется $p > 0$ байтов для дополнения для сообщения и MAC, используется $p - 1$ случайный байт, а последний байт устанавливается в значение $(p - 1)$. Если сообщение уже необходимой длины – добавляется новый блок.
- Шифрование вычисляется на дополненном открытом тексте и MAC

Атака на SSL 3.0 (предполагая случайный IV)

Пусть противник получил некоторый шифртекст $c = E((k_e, k_m), m)$ для некоторого неизвестного сообщения m. Пусть длина сообщения такова, что сообщение и MAC дополняются полным блоком дополнения. Тогда шифртекст выглядит следующим образом:



Противник создаёт новый шифртекст c', заменяя последний блок на $c[1]$



Атака на SSL 3.0 (предполагая случайный IV)

При расшифровании последнего блока получатель имеет:

$$v = D(k_c, c[1]) \oplus c[l-1] = m[0] \oplus c[0] \oplus c[l-1]$$

Если последний байт равен 15, то весь последний блок будет отброшен как дополнение. Оставшаяся часть отрытого текста образует корректную пару открытый текст – MAC и сервер не сообщит об ошибке.

Если последний байт не равен 15, то часть последнего блока будет интерпретироваться как MAC, в результате сервер вернёт $\perp$.

Атака на SSL 3.0 (предполагая случайный IV)

Итого, если сервер не вернул $\perp$, тогда противник узнаёт, что последний байт $m[0]$ равен последнему байту $u = 15 \oplus c[0] \oplus c[l-1]$. Таким образом противник вычисляет байт открытого текста и нарушает семантическую стойкость.

Атаки данного типа носят называние padding oracle attack – т.е. имея оракул дополнения, который сообщает противнику корректно ли дополнение, противник осуществляет атаку.

Реальная атака на SSL 3.0

Пусть пользователь использует веб-браузер для работы с сайтом банка, использующем SSL 3.0. После аутентификации сайт банка выдаёт пользователю cookie, которую он используется для дальнейшей аутентификации своих действий. Для этого пользователь прикладывает cookie во всех своих запросах, например:

```
GET path  cookie: cookie
```

Cookie должна оставаться секретной. Секретность обеспечивается только SSL.

Реальная атака на SSL 3.0

Цель противника – восстановить cookie из шифртекста. Противник используется межсайтовый скриатинг (XSS) или плагин браузера для отправки запросов от имени пользователя. Браузер пользователя отправляет запрос вида

```
GET /AA  cookie: cookie
```

Зашифрованный SSL. Противник перехватывает шифртекст, использует атаку, описанную ранее, и восстанавливает последний байт cookie.

Затем противник заставляет браузер пользователя отправить запрос с телом, на один байт длиннее предыдущего, например:

```
GET /AAA  cookie: cookie
```

После чего он получает в одном из блоков cookie, сдвинутую на 1 байт вправо, и восстанавливает второй байт отправляя данный блок в качестве последнего блока шифртекста.

TLS 1.0

TLS 1.0 исправил проблему дополнения – теперь все байты дополнения должны быть равны $p - 1$ (данный подход используется до сих пор).

Но реализация всё равно уязвима к одной из вариаций padding oracle – timing padding oracle.

TLS 1.0 Расшифрование

Расшифрование производится следующим образом:

- CBC расшифрование шифртекста
- Проверка дополнения, если не корректен – ошибка
- Проверка MAC, если не корректен – ошибка

Проверка MAC производилась только при корректности дополнения.

Timing padding oracle

Пусть противник имеет некоторый шифртекст c некоторого сообщения m. Пусть противник хочет проверить, является ли последний байт $m[2]$ равным некоторой величине b. Пусть B произвольный 16 байтный блок, последний блок которого равен b.

Противник создаёт новый блок $c'[1] = c[1] \oplus B$ и отправляет шифртекст $c' = (c[0], c'[1], c[2])$ серверу.

После расшифрования сервером последний блок шифртекста равен $m'[2] = c'[1] \oplus D(k, c[2]) = m[2] \oplus B$.

Если последний байт $m[2]$ равен b, тогда $m[2]$ закончится 0 – корректными дополнением и сервер начнёт проверку MAC. Иначе – сервер вернёт ошибку даже не начав проверку MAC.

Timing padding oracle

Таким образом, противник, замеряя время ответа от сервера, может получить информацию о последнем байте интересующего его блока, что ломает семантическую стойкость шифра.

Бесплатно получили проблему необходимости константного времени.

Получить остальные байты сообщения можно использовав метод, описанный ранее – меняя длину открытого текста, сдвигая тем самым интересующий нас блок открытого текста (например cookie).

На самом деле – всё было ещё хуже. Сервер явно отвечал сообщениями bad_record_mac и decryption_failed.

Yet Another Padding Oracle in OpenSSL CBC Ciphersuites

- Рассмотрим уязвимость в реализации TLS, дающую уязвимость в виде возможности padding oracle (CVE-2016-2107, LuckyNegative20)
- Уязвимость в OpenSSL, использующем AES-CBC с аппаратным вычислением (AES-NI) криптографических операций (исправлено в актуальной версии)
- Все использующие данную конфигурацию на старых версиях OpenSSL уязвимы
- Трудно реализуема на практике
- Уязвимость появилась при исправлении другой уязвимости (Lucky13)

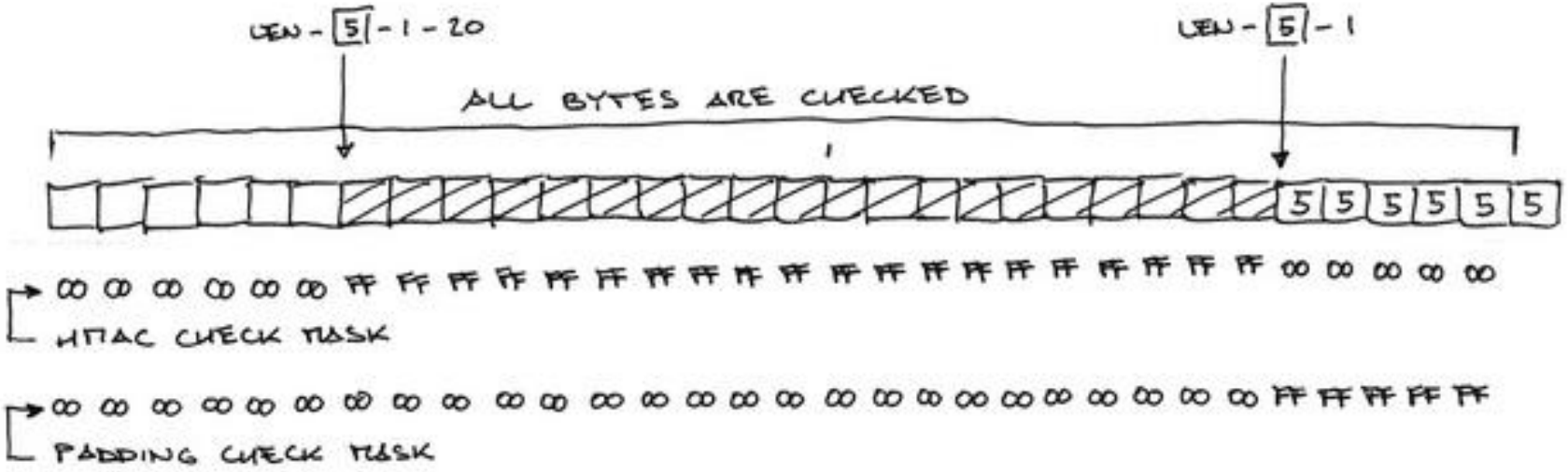
Решение проблемы константного времени

Решение проблемы константного времени – не используем if, используем AND. Вычисляем ряд значений, вычисляем AND от их результатов и возвращаем его.

Как отделить MAC от открытого текста при проверке MAC? Используется маска, накладываемая на открытый текст, показывающая, какие байты необходимо проверить.

Решение проблемы константного времени

Пример. Пусть используется HMAC (20 байт). Пусть сообщение длины 32 байта. Дополнение может быть не больше 32-1-20=11 байт. На основе длины сообщения, длины MAC и длины дополнения вычисляется маска, производится проверка дополнения, проверка MAC, вычисляется AND от результата.



Длина дополнения

```
pad = plaintext[len - 1];
maxpad = len - (SHA_DIGEST_LENGTH + 1);
maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
maxpad &= 255;
```

- maxpad – максимально возможная длина дополнения

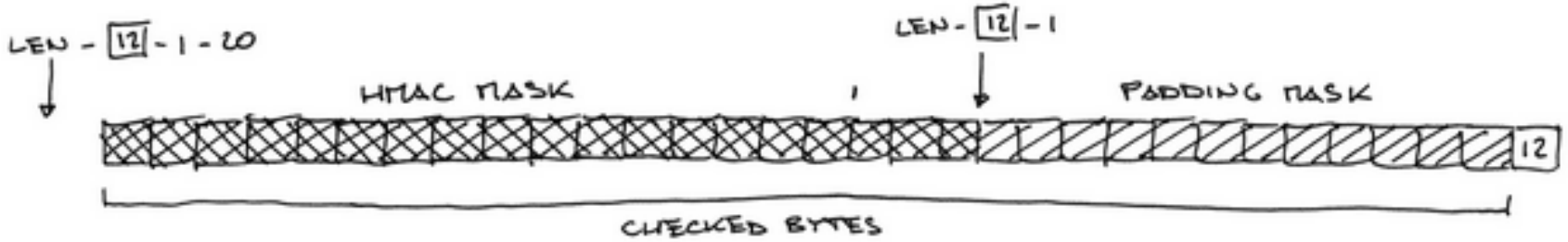
Пусть мы посылаем сообщение с дополнением

```
pad = maxpad + 1 = (len – 20 – 1) + 1 = (32 – 20 – 1) + 1 = 12
```

Длина дополнения

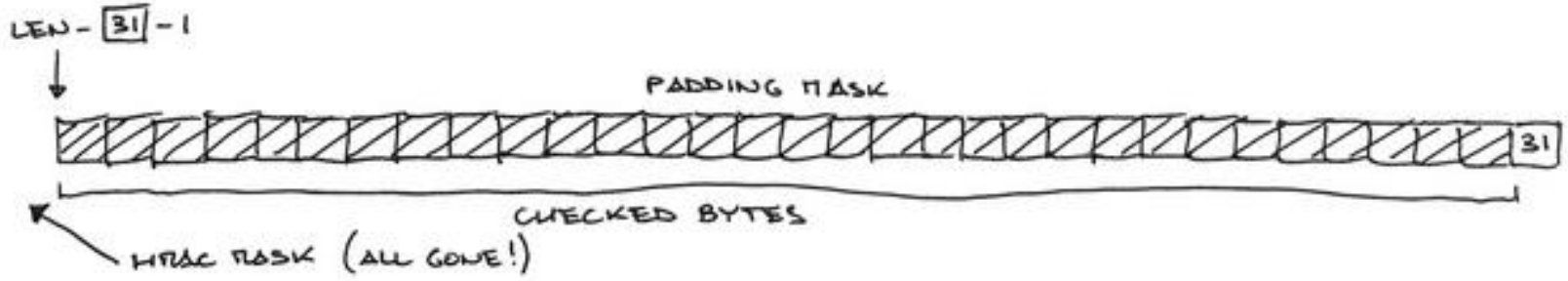
```
pad = maxpad + 1 = (len - 20 - 1) + 1 = (32 - 20 - 1) + 1 = 12
```

После расшифрования и вычисления маски:



Вычисление маски

Если задать дополнение длины 31 байт:

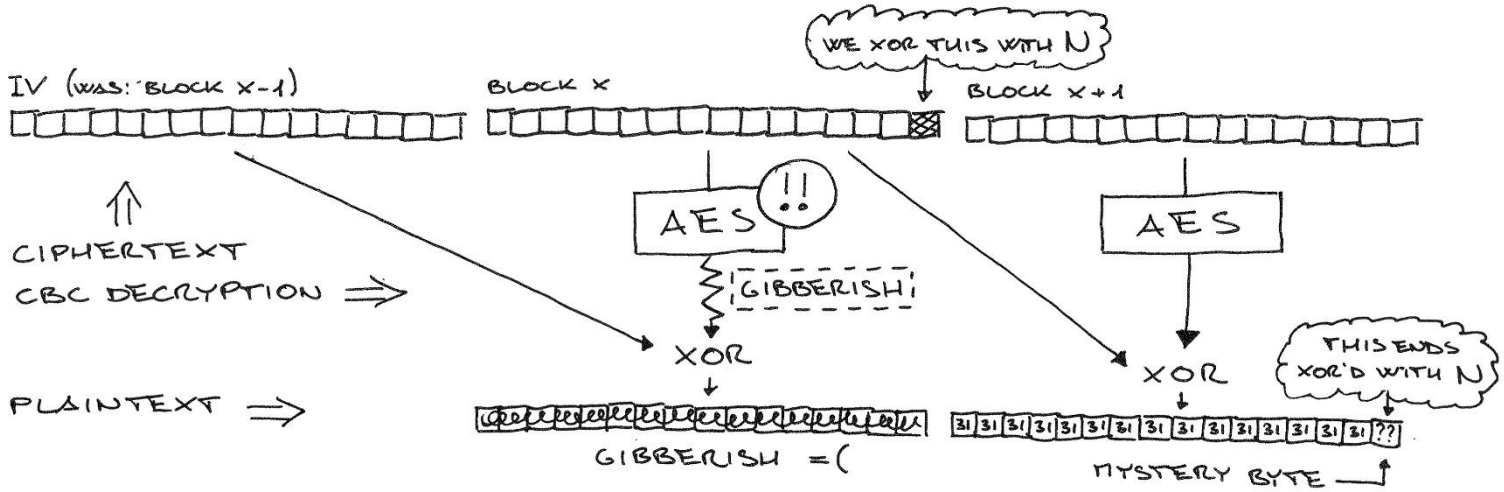


Маска для проверки MAC стала равна 0, т.е. MAC всегда коррекнтный. Проверка дополнения завершиться успешно, если все расшифрованные байты равны 31.

Возможность для атаки

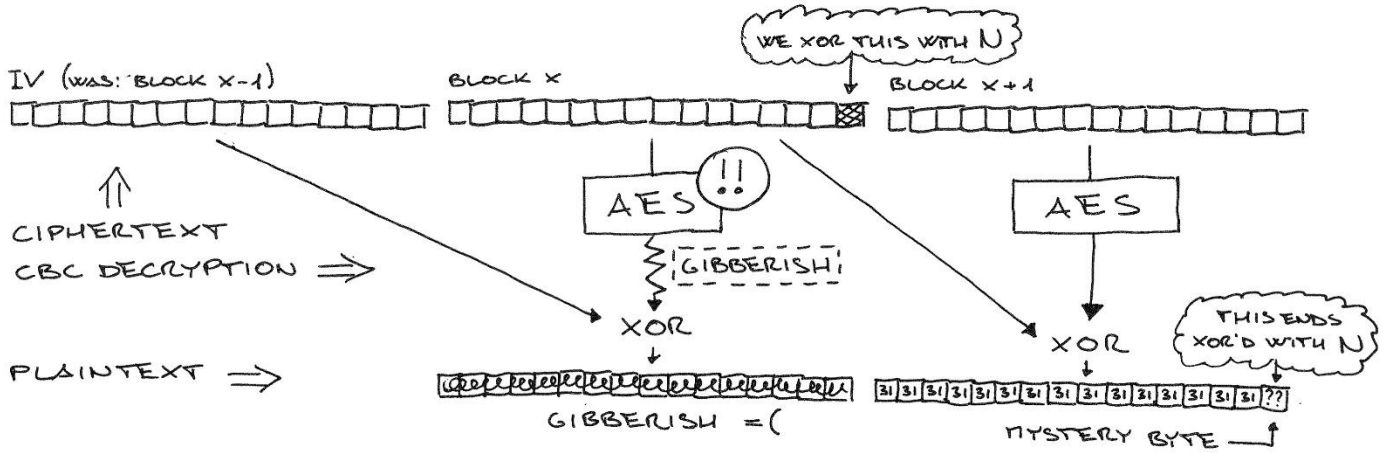
- Противник может выяснить, состоят ли сообщение из байтов n, где $n \geq \mathrm{maxpad} + 20$ отправляя его серверу и ожидая ответ отличный от BAD_MAC. Работает для сообщений не более 256-20=236 байт.
- Пусть противник контролирует префикс открытого текста, который он хочет узнать (например противник может заставлять клиента отправлять сообщения вида path || cookie, контролируя path).

Вариант атаки 1



- Противник создаёт шифртекст из трёх блоков, в котором ему не известен последний байт последнего блока открытого текста, а байты открытого текста до него имеют значения 31. Противник ксорит последний байт второго блока со значением N.
- Но, увы, не работает. Так как первый блок открытого теста состоит из случайного «мусора», полученного при расшифровании исправленного блока шифртекста. Представим пока, что при расшифровке получим всегда 31. Решим эту проблему позже в атаке 2.

Вариант атаки 1



Противник ожидает, когда сервер ответит что то кроме bad_mac, и следовательно последний байт открытого текста равен $x = u \oplus N = 31$, где u искомый байт.

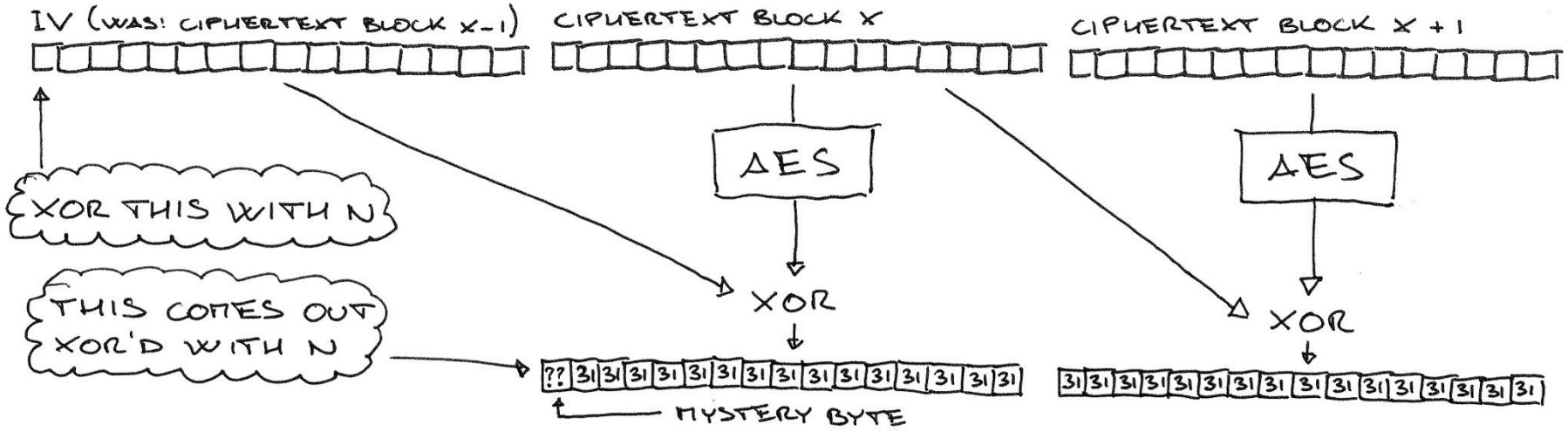
Далее противник сдвигает открытый текст влево перед зашифрованием и находит второй байт (подксоривая второй байт с конца во втором блоке найденной величиной N) на находит величину N_2 (которую необходимо подксорить к новому последнему байту во втором блоке для получения значения 31 в последнем байте отрытого текста).

Вариант атаки 2

Решим проблему «мусора» первый атаки. Создадим атаку 2.

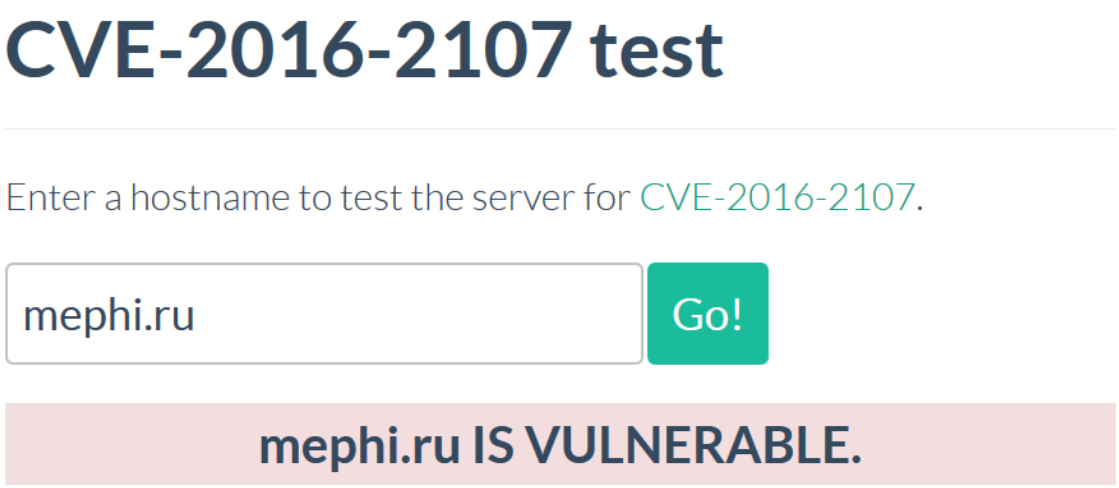
Противник контролирует постфикс открытого текста. Постфикс состоит из двух блоков, состоящих из байтов 31.

Аналогично атаке 1, ждём ответа отличного от bad_mac и восстанавливаем u: $x = u \oplus N = 31$



Проверка на уязвимость

- Запросить у сервера сообщение, расшифруемое как «AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA». Если ответ «DATA_LENGTH_TOO_LONG» - сервер уязвим. Если «BAD_RECORD_MAC» - защищен.
- А выбрана только потому, что A>32-1



Исправление ошибки

Проверка, что дополнение не выходит за разрешенные границы максимального размера дополнения.

```
            pad = plaintext[len - 1];
            maxpad = len - (SHA_DIGEST_LENGTH + 1);
            maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
            maxpad &= 255;

+           ret &= constant_time_ge(maxpad, pad);
+
```

Выводы

- Лучше использовать Encrypt-Then-MAC или один из стандартов AEAD шифрования
- Encrypt-and-MAC и MAC-then-Ecnrypt ведут к потенциальным уязвимостям в реализации и проектировании
- Никогда не придумывать криптографию
- Никогда не реализовывать криптографию