нетология

Элементы безопасности информационных систем

Андрей
Вахутинский

Андрей Вахутинский

Зам.начальника ИТ отдела в АО "ИНТЕКО"

Андрей Вахутинский

План модуля

1. Работа в терминале, лекция 1
2. Работа в терминале, лекция 2
3. Операционные системы, лекция 1
4. Операционные системы, лекция 2
5. Файловые системы
6. Компьютерные сети, лекция 1
7. Компьютерные сети, лекция 2
8. Компьютерные сети, лекция 3
9. **Элементы безопасности информационных систем**

Предисловие

Безопасность – широчайшая тема. Мы не касаемся на этой лекции:

- общей теории (свойства информации),
- классификации уязвимостей и атак,
- уязвимостей канального, сетевого, транспортного и прикладного уровней, таких как ARP-спуфинг, DNS amplification, XSS/CSRF и т.д.,
- безопасности уровня хоста (SELinux, AppArmor, POSIX ACL и т.д.),
- безопасности уровня сети (firewall – межсетевые экраны и т.д.).

Мы посмотрим на **практические моменты в деятельности** системного администратора и DevOps-инженера:

- SSH,
- PKI (Public Key Infrastructure),
- TLS в применении к HTTPS.

Рекомендуем для получения лучших базовых знаний ознакомиться с прекрасным документом за авторством Владимира Иванова.

План занятия

1. Концепции криптографии
2. TLS, Transport Layer Security
3. easy-rsa
4. SSH, Secure SHell
5. Итоги
6. Домашнее задание

Концепции криптографии

Проблема передачи открытого текста

На прошлой лекции мы наблюдали, что передавать открытый текст по сети – небезопасно. Обладая минимальными знаниями, трафик можно перехватить, вытащить из него необходимые данные (пароли, параметры платежных карт и другую конфиденциальную информацию), или даже подменить:

```
IP6 ::1.54108 > ::1.3000: Flags [P.], seq 1:191, ack 1, win 512, options [nop,...
length 190
`.wm...@.....................................\....!... ............
...M...LPOST /auth HTTP/1.1
Host: localhost:3000
User-Agent: curl/7.68.0
Accept: */*
Content-Type: application/json
Content-Length: 55

{"username":"roman", "password":"not_very_strong_pass"}
```

Выше приведен пример с протоколом HTTP, но в равной степени подобным проблемам подвержены и любые другие протоколы, работающие в незащищенных каналах: DNS, управление удаленным хостом по telnet и т.д.

Безопасность на прикладном уровне

Зная, что протоколы в модели TCP/IP заменяемы, можно предположить, что организовать защищенную передачу данных можно на разных уровнях.

На практике так и происходит: например, работающий на уровне L3 IPSec популярен в организациях.

Однако чаще всего системные администраторы и DevOps инженеры имеют дело *с публичными сервисами*, с которыми пользователи взаимодействуют по открытым каналам. Значит, полагаться на подконтрольную среду и низкоуровневые механизмы нельзя и в общем случае требуется обеспечить безопасность *на уровнях выше транспортного*.

На практике это означает, что почти у любого L7 протокола, созданного во время, когда о безопасности не задумывались так как сегодня, существует версия, обеспечивающая защищенную передачу данных:

- HTTP**S** (HTTP **S**ecure) для HTTP,
- DNS**Sec** (DNS **Sec**ure) для DNS,
- SMTP**S** (SMTP **S**ecure) для SMTP и т.д.

Достигается это использованием криптографических библиотек и зачастую общих принципов на уровнях между транспортным и прикладным.

Шифрование

Что значит передать информацию безопасно? Одно из требований:

- при передаче по открытому каналу и перехвате сторонним лицом, данные должны оставаться конфиденциальными для участников обмена.

Для обеспечения такого требования мы можем:

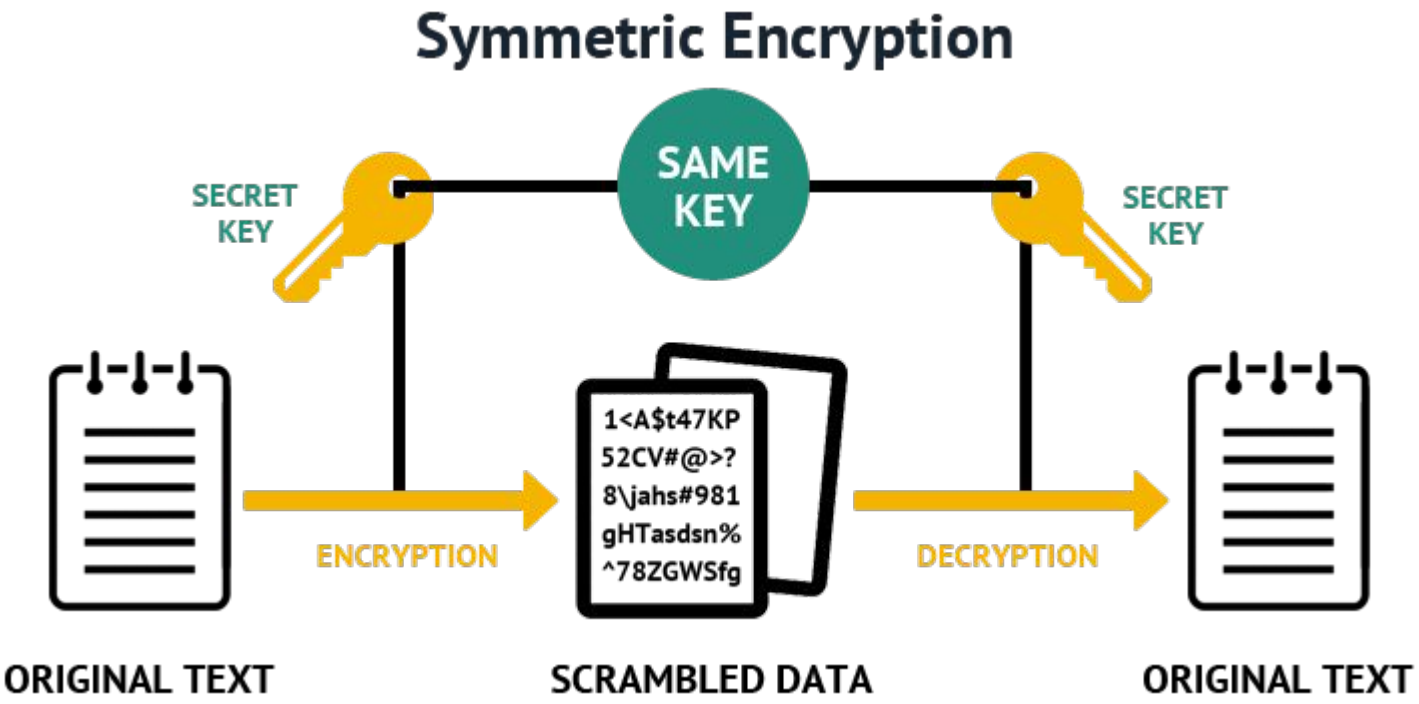
- произвести обратимые математические операции над данными (**шифрование**),
- результат которых не будет иметь смысл сам по себе (= будет **пригоден для передачи по открытому каналу**),
- без обратного преобразования (**расшифрования**), возможного только инициаторами обмена.

Существуют принципиально разные типы шифрования, которые, в свою очередь, реализованы разными алгоритмами.

Симметричное шифрование, схема

Способ, при котором шифрование и расшифрование данных происходит с использованием единого ключа, называется *симметричным*. Некоторые его современные представители:

- AES (Advanced Encryption Standard),
- ChaCha/Salsa20.



взято с сайта: cheapsslshop.com

Симметричное шифрование на примере AES

```
root@netology1:~# echo 'This is Netology DEVSYS course InfoSec webinar' > secret
root@netology1:~# gpg --output secret.encrypted --symmetric --cipher-algo AES256 secret

root@netology2:~# nc -l -p 5000 -q 1 > secret.encrypted < /dev/null
root@netology1:~# cat secret.encrypted | nc 172.28.128.60 5000

root@netology2:~# gpg --output secret --decrypt secret.encrypted
```

В данном случае для шифрования мы применили AES с длиной ключа 256 бит, **ключ** при этом был создан утилитой gpg на базе короткой фразы.

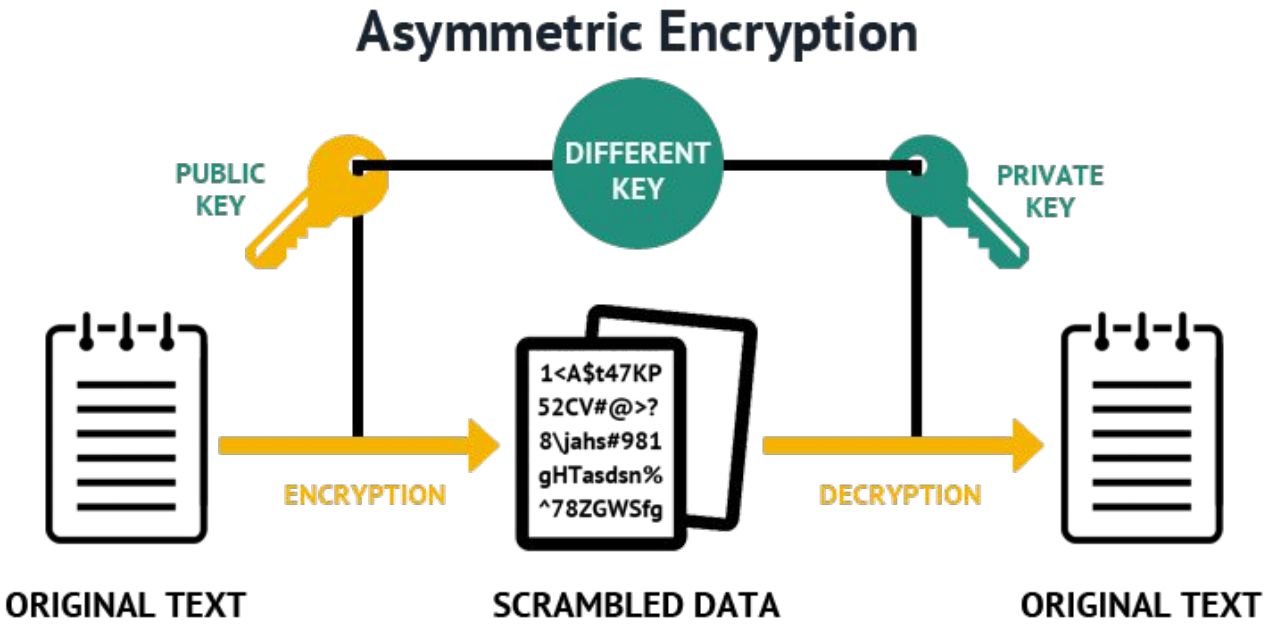
Очевидная проблема данного класса шифров – не снимается вопрос о безопасной передаче самого **ключа** между участниками обмена, поэтому одно только лишь симметричное шифрование не может обеспечить нам безопасность передачи информации по открытым каналам.

Асимметричное шифрование, схема

Проблема передачи ключа решается ассиметричным шифрованием.
Тогда как в симметричном шифровании мы используем *один и тот же ключ* для шифрования и дешифрования, в ассиметричном мы используем 2 разных (но связанных) ключа: **открытый** для шифрования и **закрытый** – для расшифрования.

Некоторые **алгоритмы для асимметричного шифрования**:

- RSA (аббревиатура от фамилий Rivest, Shamir и Adleman)
- обмен ключами DH (Diffie–Hellman, также фамилии разработчиков)



взято с сайта: cheapsslshop.com

Асимметричное шифрование на примере RSA

Для начала сгенерируем пару связанных между собой **закрытого** и **открытого** ключей:

```
root@netology1:~# openssl genrsa -out demo_private_key.pem 2048
root@netology1:~# openssl rsa -in demo_private_key.pem -pubout -out demo_public_key.pem
```

Открытый ключ является **публичным** – его можно свободно распространять любому числу клиентов, в нашем случае – парному хосту netology2, на котором мы можем использовать этот публичный ключ для шифрования сообщения:

```
root@netology2:~# echo 'A very secret text for Netology RSA slide' > our_secret
root@netology2:~# openssl rsautl -encrypt -inkey demo_public_key.pem -pubin -in
our_secret -out our_secret.encrypted

root@netology1:~# nc -l -p 5000 -q 1 > our_secret.encrypted < /dev/null
root@netology2:~# cat our_secret.encrypted | nc 172.28.128.10 5000
```

Передав назад зашифрованное сообщение, воспользуемся **приватным** ключом для его расшифровки на хосте netology1:

```
root@netology1:~# openssl rsautl -decrypt -inkey demo_private_key.pem -in
our_secret.encrypted -out our_secret
root@netology1:~# cat our_secret
A very secret text for Netology RSA slide
```

Хотя любой может **зашифровать** сообщение с использованием **публичного** ключа, только владелец **приватного** ключа может его расшифровать.

Достоинства и недостатки обеих схем

В данном примере мы сгенерировали пару ключей RSA-2048 для использования в ассиметричном шифровании и продемонстрировали, как можно безопасно обмениваться данными по открытому каналу с их помощью. Казалось бы, зачем вообще тогда нужно симметричное шифрование?

- **Ограничение на размер** шифруемого сообщения, **равное модулю ключа** 2048 бит в нашем случае
- **Несравнимая производительность**: RSA требует намного больше вычислительных мощностей несмотря на ограничение по размеру шифруемого сообщения, разница в скорости работы AES-256 даже без аппаратного ускорение (AES-NI) и RSA-2048 – **2 порядка**

Как результат – на практике асимметричное и симметричное шифрование используется **совместно**:

1. ***симметричное*** шифрование применяется для самих передаваемых данных,
2. *ассиметричное* шифрование применяется для шифрования **сессионного** ключа, который используется в **симметричном** шифровании.

Упомянутая связка RSA + AES является часто встречающейся парой, однако есть альтернативы. Например обмен ключами RSA можно заменить на ECDHE – обмен эфемерными ключами по алгоритму Диффи-Хеллмана на эллиптических кривых.

Хеширование

На предыдущих слайдах мы рассматривали результат работы обратимых математических операций – *шифрования*. Однако, полезными нам окажутся и функции другого класса – функции *хеширования*.

Если в шифровании практическая цель – безопасно передать по открытым каналам секретные данные, хеширование поможет быстро определить, являются ли сравниваемые данные одинаковыми. Для этого хеш-функция, являющаяся в отличие от шифрования **необратимой** (математически это утверждение неверно, но затраченное время на поиск должно быть очень велико для подбора), вычисляет от исходных данных **произвольного** размера строку **фиксированной** длины.

```
root@netology1:~# echo 'I am a secret string' | sha256sum
0efbe9516c9fcd85bd068874df8c44344583d80afaa0a22882d14149da0f59f4  -

root@netology1:~# echo -n 'I am a secret string' | sha256sum
ad6451476c85f8d25cd33a8ca4904b2f07fda686aa7d04a6215a6ad574583fc9  -
```

Для надежного использования в криптографии хеш-функции также должны быть стойкими к разнообразным **коллизиям** (примеры коллизий в MD5), таковыми являются:

- SHA-256/384/512 (Secure Hash Algorithms).

Цифровая подпись, целостность

Хеширование само по себе не выглядит полезным. Пускай, мы можем многократно получать один и тот же хеш от данных, при этом свести к минимуму вероятность коллизий, но какую пользу это дает? Хеширование используется в *цифровой подписи*.

Предположим, что сервер **Alice** передает клиенту **Bob** данные. На этот раз, в самих данных нет ничего секретного, однако клиент **B** хочет удостовериться, что данные были получены от сервера **A в целостности**: их никто не модифицировал по пути.

Для этого:

1. Сервер **A** создает хеш от передаваемых данных.
2. Сервер **A** подписывает хеш своим приватным ассиметричным ключом.
3. Сервер **A** передает клиенту **B**: *данные*, *хеш данных*, *публичный ключ*.
4. Клиент **B** использует публичный ключ сервера **A** для дешифровки хеша, после чего сравнивает полученный результат с собственным вычислением хеша.
5. Если результат совпадает – для клиента **B** становится известно, что при передаче между **A** и **B** **сообщение не было изменено.**

Сертификат, аутентификация

В примере с цифровой подписью мы научились важному свойству – проверять целостность доставленного сообщения. Однако, нерешенной осталась последняя важная проблема – как идентифицировать, что представляющийся **Alice** сервер действительно является тем, за кого себя выдает? Ведь *любой* может сгенерировать хеш от документа, подписать его своим закрытым ключом и отправить **Bob**, представившись **Alice**.

Получается, что нужен какой-то дополнительный признак, по которому можно будет однозначно связать *имя Alice* и открытый ключ, который *представляющийся Alice сервер* предлагает для проверки цифровой подписи.

Таким признаком является **сертификат** от доверенного CA – certification authority, или центра сертификации.

К сожалению, на данный момент без элемента доверия третьей стороне, аутентифицировать участника безопасного соединения невозможно.

Сертификат выписывается CA в ответ на csr после проверки разного уровня сложности (владение доменом через TXT запись DNS, HTTP ответ по заданному URL и т.д.) – запрос, включающий в себя ряд полей, среди которых:

- subject (например, Common Name – домен),
- информация о CA (для проверки сертификата),
- публичный ключ.

TLS, Transport Layer Security

Шифрование + аутентификация + целостность

Мы научились генерировать сессионный ключ для симметричного шифрования и обмениваться им благодаря криптографии с публичными ключами – асимметричному шифрованию. Поставленная изначально **задача передачи по открытым каналам секретного сообщения**, казалось бы, решена.

Благодаря **цифровой подписи** публичного асимметричного ключа мы гарантируем, что при передаче публичного ключа между участниками обмена он не был модифицирован по пути.

Наконец, так как сервер предоставил не только подписанный публичный ключ, но и сертификат, выпущенный доверенным CA, мы можем **аутентифицировать** сервер, подтвердив соответствие публичного ключа и имени сервера.

Только когда все эти условия будут соблюдены, можно считать, что зашифрованное соединение установлено.

Набор стандартов, который определяет допустимые протоколы для каждого из этих этапов, включая размерности ключей, возможность сочетания разных алгоритмов шифрования, хеширования и подписи, называется TLS или Transport Layer Security.

Стандарт является развитием SSL (устаревшего на сегодня даже в самой старшей версии), и постоянно актуализируется в соответствии с угрозами и новыми типами атак. Рекомендации Mozilla.

Распространенные современные стандарты TLS

```
nmap --script ssl-enum-ciphers -p 443 ssllabs.com
PORT    STATE SERVICE
443/tcp open  https
| ssl-enum-ciphers:
|   TLSv1.2:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (ecdh_x25519) - A
|       TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (dh 2048) - A
…

nmap --script ssl-enum-ciphers -p 443 ya.ru
|   TLSv1.2:
|     ciphers:
|       TLS_RSA_WITH_AES_128_CBC_SHA256 (rsa 2048) - A
```

Посмотреть доступные локально надежные алгоритмы (пример):

```
openssl ciphers -v 'ECDHE+AES256' | column -t
ECDHE-RSA-AES256-GCM-SHA384    TLSv1.2  Kx=ECDH  Au=RSA    Enc=AESGCM(256)  Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384  TLSv1.2  Kx=ECDH  Au=ECDSA  Enc=AESGCM(256)  Mac=AEAD
ECDHE-RSA-AES256-SHA384        TLSv1.2  Kx=ECDH  Au=RSA    Enc=AES(256)     Mac=SHA384
ECDHE-ECDSA-AES256-SHA384      TLSv1.2  Kx=ECDH  Au=ECDSA  Enc=AES(256)     Mac=SHA384
```

Где Kx – **K**ey e**X**change, ассиметричная генерация сессионного ключа,
Au – **Au**thentication, цифровая подпись,
Enc – **Enc**ryption, симметричное шифрование сессионным ключом,
Mac – **M**essage **a**uthentication **c**ode, проверка целостности сообщения хеш-функцией.

Некоторые алгоритмы симметричного шифрования, например, AES GCM, являются AEAD алгоритмом и не требуют отдельно Mac.

easy-rsa

Набор скриптов для автоматизации CA

В прошлом разделе мы на практике посмотрели работу некоторых алгоритмов шифрования, воспользовавшись gpg и openssl. Возможностей openssl хватает для организации собственного CA, однако необходимость указания большого числа ключей для работы с утилитой делает затруднительным ее прямое использование для такой задачи в реальных условиях. Один из вариантов решения такой задачи – пакет easy-rsa от создателей OpenVPN.

```
root@netology3:~# apt install -y easy-rsa
root@netology3:~# make-cadir demo_ca; cd $_
root@netology3:~/demo_ca# ls -l
total 20
lrwxrwxrwx 1 root root   27 08:58 easyrsa -> /usr/share/easy-rsa/easyrsa
-rw-r--r-- 1 root root 4651 08:58 openssl-easyrsa.cnf
-rw-r--r-- 1 root root 8576 08:58 vars
lrwxrwxrwx 1 root root   30 08:58 x509-types -> /usr/share/easy-rsa/x509-types
root@netology3:~/demo_ca# grep set_var vars ...
```

Здесь мы создали директорию для нашего будущего демонстрационного CA. Ознакомьтесь с настройками по умолчанию в vars, которые будут задействованы в процессе работы с сертификатами, такими как:

- EASYRSA_REQ_COUNTRY
- EASYRSA_REQ_OU
- и т.д.

Инициализация CA

Значения по-умолчанию из vars следует заменить, но для целей демонстрации это делать необязательно. Инициализируем PKI для CA:

```
# подготовит конфигурационный файл openssl
root@netology3:~/demo_ca# ./easyrsa init-pki
# сгенерирует пару PKI ключей CA,
# csr на самоподписанный сертификат для CA, подпишет его
root@netology3:~/demo_ca# ./easyrsa build-ca nopass
root@netology3:~/demo_ca# openssl x509 -in pki/ca.crt -text
```

Инициализируем PKI для клиента, который сгенерирует csr на сертификат для домена example.com. Этот csr мы импортируем в CA для последующей подписи:

```
root@netology3:~# cd ~; make-cadir tls_server1; cd $_
root@netology3:~/tls_server1# ./easyrsa init-pki
root@netology3:~/tls_server1# ./easyrsa gen-req example.com nopass
Keypair and certificate request completed. Your files are:
req: /root/tls_server1/pki/reqs/example.com.req
key: /root/tls_server1/pki/private/example.com.key
```

Обрабатываем импортированный csr, получая на выходе подписанный приватным ключом CA серверный сертификат, который мы уже можем использовать:

```
root@netology3:~/demo_ca# ./easyrsa import-req
/root/tls_server1/pki/reqs/example.com.req example.com
root@netology3:~/demo_ca# ./easyrsa sign-req server example.com
Certificate created at: /root/demo_ca/pki/issued/example.com.crt
```

Информация о csr и crt

С помощью openssl можно посмотреть **информацию как о csr**:

```
root@netology3:~# openssl req -in tls_server1/pki/reqs/example.com.req -noout -pubkey
-----BEGIN PUBLIC KEY-----
…
root@netology3:~# openssl req -in tls_server1/pki/reqs/example.com.req -noout
-subject
subject=CN = example.com
```

Так и **о выданном сертификате**:

```
root@netology3:~# openssl x509 -in demo_ca/pki/issued/example.com.crt -noout -dates
notBefore=Aug  3 09:08:30 2020 GMT
notAfter=Jul 19 09:08:30 2023 GMT

root@netology3:~# openssl x509 -in demo_ca/pki/issued/example.com.crt -noout -subject
subject=CN = example.com

root@netology3:~# openssl x509 -in demo_ca/pki/issued/example.com.crt -noout -issuer
issuer=CN = Easy-RSA CA
```

Обратите внимание, что в обсуждаемом примере только сервер предоставляет сертификат для свой идентификации, однако TLS этим не ограничивается, можно при необходимости, например, проводить аутентификацию с обеих сторон (mutual TLS authentication).

CN vs DNS Alternative Names – не используйте CN!

Использование сертификата в nginx

Простейший вариант демонстрации успешного использования полученного сертификата: добавить его к vhost по умолчанию в веб-сервер nginx:

```
root@netology3:~# vim /etc/nginx/sites-enabled/default
# раскомментируем строки:
        # listen 443 ssl default_server;
        # listen [::]:443 ssl default_server;
# добавим пути до приватного ключа и сертификата с публичным ключем
        ssl_certificate /root/demo_ca/pki/issued/example.com.crt;
        ssl_certificate_key /root/tls_server1/pki/private/example.com.key;
root@netology3:~# nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
root@netology3:~# systemctl reload nginx
root@netology3:~# systemctl status nginx
```

Мы выписали сертификат для example.com, сделаем теперь этот домен “доступным” на localhost и попробуем обратиться к нему:

```
root@netology3:~# echo 127.0.0.1 example.com >> /etc/hosts
root@netology3:~# host example.com
example.com has address 127.0.0.1
root@netology3:~# curl -I https://example.com
curl: (60) SSL certificate problem: unable to get local issuer certificate
```

Почему так происходит? Хотя мы и выписали сертификат для домена на нашем хосте, это было сделано в одной из папок, принадлежащих root. ОС ничего не знает о том, что CA, который мы создали easy-rsa должен считать доверенным.

Просмотр системного trust-store

В дистрибутиве ОС распространяется и набор корневых доверенных сертификатов ЦА:

```
root@netology3:~# ls -l /etc/ssl/certs
```

Чтобы добавить наш вновь созданный сертификат ЦА в доверенные:

```
root@netology3:~# ln -s /root/demo_ca/pki/ca.crt
/usr/local/share/ca-certificates/demo_ca.crt
root@netology3:~# update-ca-certificates
Updating certificates in /etc/ssl/certs...
1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
```

Таким образом, приложения, которые пользуются системным хранилищем доверенных сертификатов (curl входит в их число), будут принимать сертификаты, подписанные demo_ca:

```
root@netology3:~# curl -I -s https://example.com | head -n1
HTTP/1.1 200 OK
```

Следует отметить, что существует большое количество приложений, которые не пользуются системных хранилищем. Среди них – виртуальная машина Java, браузер Firefox и т.д.

CRL и OCSP

Две модели для проверки валидности сертификата.

Certificate **R**evocation **L**ist, список серийных номеров отозванных сертификатов:

```
openssl s_client -connect ya.ru:443 2>/dev/null | openssl x509 -noout -text 2>&1 |
grep 'X509v3 CRL Distribution Points' -A3
            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://crls.yandex.net/certum/ycasha2.crl

openssl crl -inform DER -text -noout -in ycasha2.crl
```

Online **C**ertificate **S**tatus **P**rotocol, проверка сертификата в реальном времени:

```
openssl s_client -connect ya.ru:443 -showcerts 2>&1 < /dev/null

openssl ocsp -issuer chain.pem -cert ya.ru.pem -text -url https://ocsp.certum.pl

OCSP Response Data:
    OCSP Response Status: successful (0x0)
```

SSH, Secure SHell

Публичный + приватный ключ пользователя

Посмотреть конфигурацию vagrant ssh:

```
rg@m:~/netology/Vagrant$ vagrant ssh-config
Host netology1
  HostName 127.0.0.1
  User vagrant
  Port 2222
  IdentityFile /Users/rgershkovich/netology/Vagrant/../virtualbox/private_key
```

Воспользоваться обычным ssh клиентом вместо vagrant ssh:

```
ssh vagrant@localhost -i
/Users/rgershkovich/netology/Vagrant/.vagrant/machines/netology1/virtualbox/private_k
ey -p 2222
```

Сгенерировать новую пару ключей:

```
ssh-keygen -t ed25519 -C "netology@example.com"
> /home/vagrant/.ssh/authorized_keys

ssh vagrant@localhost -i ./id_ed25519_demo -p 2200
```

- Используйте ssh-agent!
- Используйте ssh -vvv для максимально детальных сообщений при дебаге проблем ssh.
- Узнайте о том, как Facebook использует CA и сертификаты для ssh.

Итоги

- Познакомились с TLS и технологиями, обеспечивающими защищенные соединения: симметричной и асимметричной криптографией, хеш-функциями;
- понятием PKI – Public Key Infrastructure на примере набора скриптов easy-rsa и самоподписанного ЦА;
- о применении ключей для SSH.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Андрей Вахутинский

Андрей Вахутинский