On using Trusted Execution Environment for TLS session signing

2019-04-15

Problem description

Typically, a TLS server uses a Certificate and associated Private Key in order to sign TLS session. From now on I'll call this Private Key a "traffic- private-key". Both certificate and traffic-private-key form a asymmetric cryptographic key-pair. Revealing the traffic-private-key makes it possible to perform men-in-the-middle type of attacks. Typically traffic-private-key is stored on the server's hard disk. Even if traffic-private-key is stored in encrypted form, at some point HTTPS server needs to have a possibility to decrypt it in order to use for signing. It means that at runtime the key in plaintext will be available in a memory of a HTTPS process. At this point attacker with an access to the machine may be able to dump memory of the process and learn the traffic-private-key.

Hence, server operators need to take special care in order to make sure traffic-private-keys are not revealed.

This situation gets more complicated in cases when server operator and domain owner are 2 different entities. For example in case of CDN, TLS offloading happens on the edge system - which often is a completely different machine than actual application server. Also it is often the case that servers (physical machines) of CDN provider are spread over the world and are located in remote data centers. Those data centers may be owned by multiple different entities.

In such situations, problem of ensuring that the traffic-private-key is not copied and used by an attacker may be challenging and not obvious to solve. Clients of the CDN may also be concerned about idea of spreading the traffic-private-key over the world.

Solution proposed

For brevity I'm assuming server uses only TLS 1.3 as specified in [RFC8446], but solution can be adapted to any version of TLS.

The idea is to perform TLS session signing inside Trusted Execution Environment. The traffic-private-key will be accessible only to TEE. Additionally, solution ensures that key is stored in encrypted form in trusted storage. The storage is bound to the physical machine and hence copy of the storage can't be used on some different machine.

The solution as implemented in the PoC and described below is based on ARM TrustZone and it uses open sourced TEE called OP-TEE (see [OP-TEE]), sources of OP-TEE are stored on github (see [OP-TEE-SRC]). OP-TEE was driven by the fact that author is quite familiar with environment nevertheless it can be implemented with other TEEs which provide device bound trusted storage. Author is convinced that Intel SGX with Asylo would be better choice here. Solution makes also heavy use of BoringSSL for handling with TLS traffic.

Points below describe implementation in more details:

1. Key provisioning server

   It is assumed that machine is initially provisioned with a software which acts as a server for traffic-private-key provisioning.

   In order to install traffic-private-key on a machine, operator connects to key provisioning server and sends the traffic-private-key to be installed on the machine. This operation is done over TLS connection which uses client authentication. Possition of some form of TLS provisioning is required by the operator. Key provisioning server must be able to verify provisioning key, hence verification-provisioning-key is also preinstalleld.

   After sucessuful TLS authentication, operator sends a pair of traffic-private-key and domain name for which the key must be used. This pair is installed on secure storage which accessible from TEE only. TEE ensures traffic-provisioning-key can't be read from outside of TEE.

2. TLS session signing

   Solution uses BoringSSL to offload TLS traffic. BoringSSL API gives a possibility to register a function which is called during TLS handshake, when server needs to sign a session with traffic-private-key.

   It means that there are no modifications needed to BoringSSL in order to use it for signing TLS session with traffic-private-key stored in TEE.

   The code which registers signing operation looks like this:

```
void signing_operation(message_to_sign, domain_name, *signature) {
    ... calls TEE for signing ...
}

SSL_PRIVATE_KEY_METHOD private_key_methods {
    .sign = signing_operation
    .decrypt = ...
```

```
    .complete = ...
};
SSL_CTX_set_private_key_method(SSL_CTX, &private_key_methods)
```

TLS server calls `signing_operation` function when TLS session needs to be signed. This function passes `message_to_sign` and `domain_name` to the TEE. While in the TEE, the `domain_name` is used as an index in order to retrieve right traffic-private-key (many domains can be handled by the server). TEE performs signing and `signature` is returned to the BoringSSL. BoringSSL continues TLS handshake as normal.

3. Key can't be used on another machine.

   Trusted storage in OP-TEE is bound to the physical device. It means even if the storage is coppied to another device, it won't be possible to decrypt stored data.

   In more detail, OP-TEE implements GlobalPlatform Trusted Storage API. Device binding is one of the requirements for trusted storage. In order to make it possible each device needs to come with preinstalled Hardware Unique Key (HUK).

   More details about trusted storage can be found on in OP-TEE documentation (see [OP-TEE-STORAGE]).

   It must be mentioned, that in order to use trusted storage, SoC specific customization is needed (see comment in orange at the bottom of [OP-TEE-STORAGE]).

PoC implementation

As mentioned before, implementation uses OP-TEE as a base for TEE. Example was tested with OP-TEE running inside QEMU emulating ARMv8.

PoC is composed of:

- `admin_cli`: Client used for installing the private keys inside TEE. This component is used instead of `key provisioning server` as such server was not implemented in PoC.
- `server`: It is a TLS offloading server. Server listens on `127.0.0.1:443` and uses BoringSSL to accept and handle TLS connection. Server implements function callback, which calls TEE when private key operation needs to be done. Only ECDSA/P256 sining is currently supported.
- `ta`: Trusted application running inside TEE. The application is responsible for processing requests from `admin_cli`, which is storing the keys on trusted storage and deleting them if requested. As well as processing signing requests from the `server`.

The section called "Example of usage" explains how to use the software in details.

Compilation and installation

Following steps need to be taken to install the software:

1. OP-TEE building. This step is explained in details here. It is required to perform steps 1 to 5. The `TARGET` (see the building instruction) used by this example is called `QEMUv8`. In case OP-TEE is started after step 5, it has to be stopped.
2. Next steps assume that Linux operating system is used and OP-TEE has been cloned to the directory called `OPTEE_DIR`.
3. Create directory `/tmp/tee_shared`
4. Go to `OPTEE_DIR` directory.
5. Clone `git clone https://git.amongbytes.com/kris/c3-tls-sign-delegator.git projects`
6. Compile BoringSSL for aarch64 and native system: `cd OPTEE_DIR/projects/bssl; make`. Makefile is configured to use toolchain build in step 1. This step will also build BoringSSL for host machine, it requires all dependencies for building BoringSSL are installed (see [BORING-BUILD]).
7. Compile solution: `cd OPTEE_DIR/projects/delegator; make`

Start process

1. Starting OP-TEE: User needs to:
   - Enter build directory: `cd OPTEE_DIR/build`
   - Start QEMU with OP-TEE emulation: `make QEMU_VIRTFS_ENABLE=y QEMU_USERNET_ENABLE=y QEMU_VIRTFS_HOST_DIR=/tmp/tee_share HOSTFWD=",hostfwd=tcp::1443-:1443" run-only`.
   - Just after qemu starts it will pause with following prompt:

```
cd /home/hdc/repos/optee/qemuv8/build/../out/bin && /home/hdc/repos/optee/qemuv8/build/.
    -nographic \
    -serial tcp:localhost:54320 -serial tcp:localhost:54321 \
    -smp 2 \
    -s -S -machine virt,secure=on -cpu cortex-a57 \
    -d unimp -semihosting-config enable,target=native \
    -m 1057 \
    -bios bl1.bin \
    -initrd rootfs.cpio.gz \
    -kernel Image -no-acpi \
    -append 'console=ttyAMA0,38400 keep_bootcon root=/dev/vda2' \
```

```
    -fsdev local,id=fsdev0,path=/tmp/tee_share,security_model=none -device virtio-9p-dev
QEMU 3.0.93 monitor - type 'help' for more information
(qemu)
```

User continues the process by entering `c`

```
(qemu) c
```

After a while 2 additional terminals should appear - one terminal labeld as “Normal”, running linux and another terminal labeled as “Secure” with output from the TEE. %3Chttps://youtu.be/02klEwlsJIA%3E

2. In the “Normal World” terminal user needs to mount file system to share data between guest and host machine. Following command needs to be used:

```
mount -t 9p -o trans=virtio host /mnt
```

%3Chttps://youtu.be/5psOtKtdlWI%3E

3. Install Trusted Application inside OP-TEE:

In the “Normal” terminal invoke:

```
sh /mnt/out/etc/tee_install
```

%3Chttps://youtu.be/aGbqgz_e9Ec%3E

At this point installation and startup process is complated and solution can be used.

Example of usage

1. Installing a key on secure storage

First step is to install the key on secure storage. Ideally this step is done by “Key provisioning server”. Nevertheless, this PoC doesn’t implement such server. Instead `admin_cli` can be used to install the key.

In the “Normal” terminal, go to `/mnt/out/` and invoke

```
cd /mnt/out
# ./admin_cli/admin_cli put www.test.com etc/ecdsa_256.key
```

This command installs private key for `www.test.com`. In the “Secure” terminal you should see a message `E/TA: install_key:156 Storing a key`. After this step `etc/ecdsa_256.key` can be removed.

%3Chttps://youtu.be/___7WKvx8XxM%3E

2. Start a TLS server and perform TLS handshake:

   With private key installed TLS server can be started. In the "Normal" terminal invoke

```
> cd /mnt/out
> ./server/server
```

   Server will start listening on `127.0.0.1:1443`. In the host machine one can try to connect to the TLS server:

```
> cd OPTEE_DIR

> ./projects/bssl/src/build.native/tool/bssl client -connect 127.0.0.1:1443 -server-name
Connecting to 127.0.0.1:1443
Connected.
Version: TLSv1.3
Resumed session: no
Cipher: TLS_AES_128_GCM_SHA256
ECDHE curve: X25519
Signature algorithm: ecdsa_secp256r1_sha256
Secure renegotiation: yes
Extended master secret: yes
Next protocol negotiated:
ALPN protocol:
OCSP staple: no
SCT list: no
Early data: no
Cert subject: CN = www.dmv.com
Cert issuer: C = FR, ST = PACA, L = Cagnes sur Mer, OU = Domain Control Validated SARL,
```

   %3Chttps://youtu.be/kRRl2zhbUqc%3E

3. Trial to access different domain fails as traffic-private-key is not available.

   %3Chttps://youtu.be/LBhllWcn4RY%3E

Extensions to the idea

First of all - key storage is bound to the device. In order to use stolen key for MITM, attacker needs to steal whole machine, which is much more difficult and easier to control. In order to implement such solution user doesn't need expensive HSM, but it can simply use Intel with SGX and Asylo. Also it's easy to imagine some extensions to this idea. For example instead of calling TEE each time for session signing during TLS handshake, one could imagine that solution can use "Delegated Credentials for TLS" (see https://tools.ietf.org/html/draft-rescorla-tls-subcerts-02). In this case TEE would be responsible for generating short lived certificates and TLS server

would request such certificate every fixed amount of time (every few minutes). This idea could be combined with another – instead of storing traffic-private-key in multiple machines, one could imagine storing a key in some central location with more restricted access (but still in TEE). Combining those two ideas improves security of traffic-private-key storage without degrading time needed to perform TLS handshake. It must be noticed that "Delegated Credentials for TLS" are already implemented in BoringSSL.

Links

- [PoC code] https://git.amongbytes.com/kris/TLS-delegation-to-TEE
- [RFC8446] https://tools.ietf.org/html/rfc8446
- [OP-TEE] https://www.op-tee.org/
- [OP-TEE-SRC] https://github.com/OP-TEE
- [OP-TEE-STORAGE] https://optee.readthedocs.io/architecture/secure_storage.html
- [BORING-BUILD] https://github.com/google/boringssl/blob/master/BUILDING.md