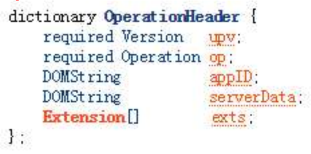
一些概念名词解释

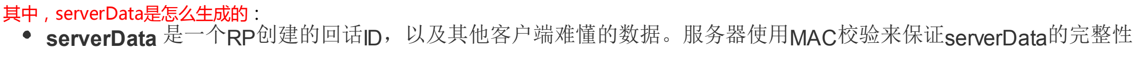
笔记本：	FIDO		
创建时间：	2015/10/27 16:15	**更新时间：**	2015/10/28 20:21
作者：	yangzhou1989@gmail.com		
URL：	https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html#operationheader-dictionary		

OperationHeader:



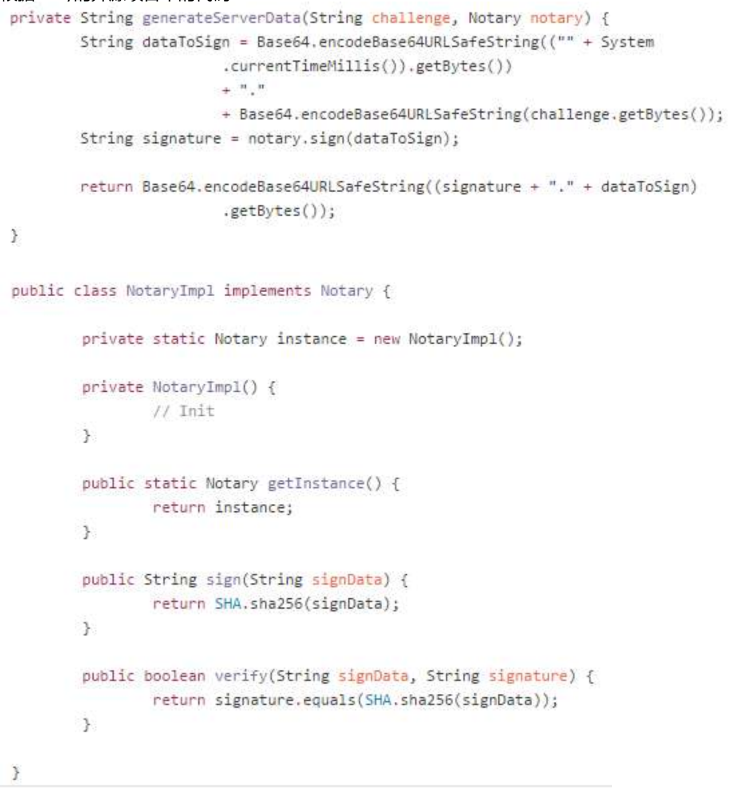
```
dictionary OperationHeader {
    required Version   upv;
    required Operation op;
    DOMString          appID;
    DOMString          serverData;
    Extension[]        exts;
};
```

其中，serverData是怎么生成的：



- **serverData** 是一个RP创建的回话ID，以及其他客户端难懂的数据。服务器使用MAC校验来保证serverData的完整性

根据ebay的开源项目中的代码：



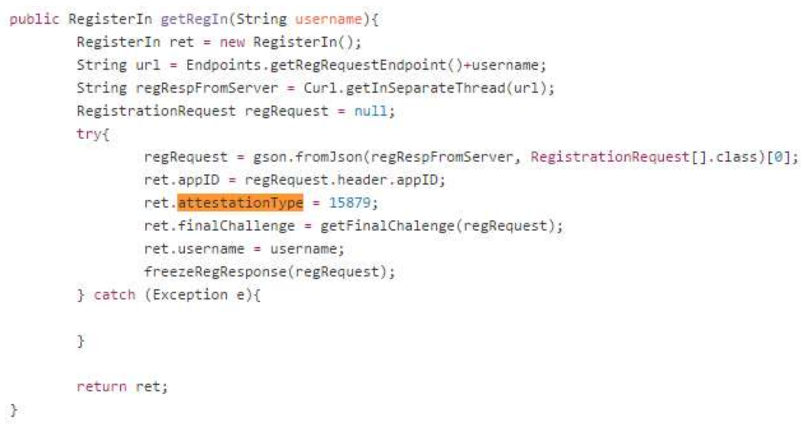
```
private String generateServerData(String challenge, Notary notary) {
        String dataToSign = Base64.encodeBase64URLSafeString(("" + System
                        .currentTimeMillis()).getBytes())
                        + "."
                        + Base64.encodeBase64URLSafeString(challenge.getBytes());
        String signature = notary.sign(dataToSign);

        return Base64.encodeBase64URLSafeString((signature + "." + dataToSign)
                        .getBytes());
}

public class NotaryImpl implements Notary {

        private static Notary instance = new NotaryImpl();

        private NotaryImpl() {
                // Init
        }

        public static Notary getInstance() {
                return instance;
        }

        public String sign(String signData) {
                return SHA.sha256(signData);
        }

        public boolean verify(String signData, String signature) {
                return signature.equals(SHA.sha256(signData));
        }

}
```

dataToSign = 系统时间+challenge
signature = SHA.sha256(dataToSign)
serverData = signature.dataToSign用Base64编码

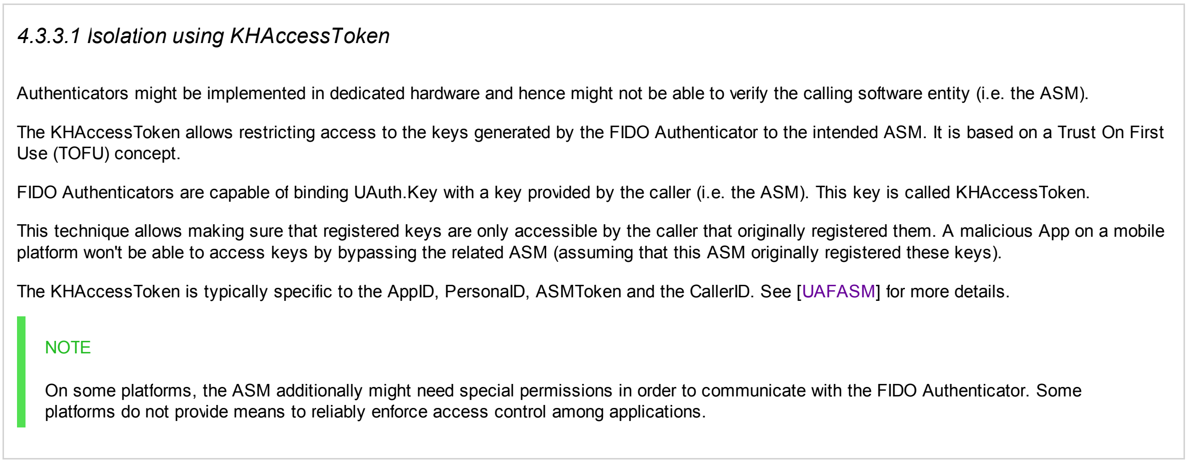
AttestationType:
attestation type supported by the authenticator. 协议中定义了3种(https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html):

TAG_ATTESTATION_CERT 0x2E05
    Indicates DER encoded attestation certificate.
TAG_ATTESTATION_BASIC_FULL 0x3E07
    Indicates full basic attestation as defined in [UAFProtocol].
TAG_ATTESTATION_BASIC_SURROGATE 0x3E08
    Indicates surrogate basic attestation as defined in [UAFProtocol].

在ebay的开源代码中能够找到这样的代码段：


```
public RegisterIn getRegIn(String username){
        RegisterIn ret = new RegisterIn();
        String url = Endpoints.getRegRequestEndpoint()+username;
        String regRespFromServer = Curl.getInSeparateThread(url);
        RegistrationRequest regRequest = null;
        try{
                regRequest = gson.fromJson(regRespFromServer, RegistrationRequest[].class)[0];
                ret.appID = regRequest.header.appID;
                ret.attestationType = 15879;
                ret.finalChallenge = getFinalChalenge(regRequest);
                ret.username = username;
                freezeRegResponse(regRequest);
        } catch (Exception e){

        }

        return ret;
}
```
其中15879就是0x3E07

KHAccessToken：


4.3.3.1 Isolation using KHAccessToken

Authenticators might be implemented in dedicated hardware and hence might not be able to verify the calling software entity (i.e. the ASM).

The KHAccessToken allows restricting access to the keys generated by the FIDO Authenticator to the intended ASM. It is based on a Trust On First Use (TOFU) concept.

FIDO Authenticators are capable of binding UAuth.Key with a key provided by the caller (i.e. the ASM). This key is called KHAccessToken.

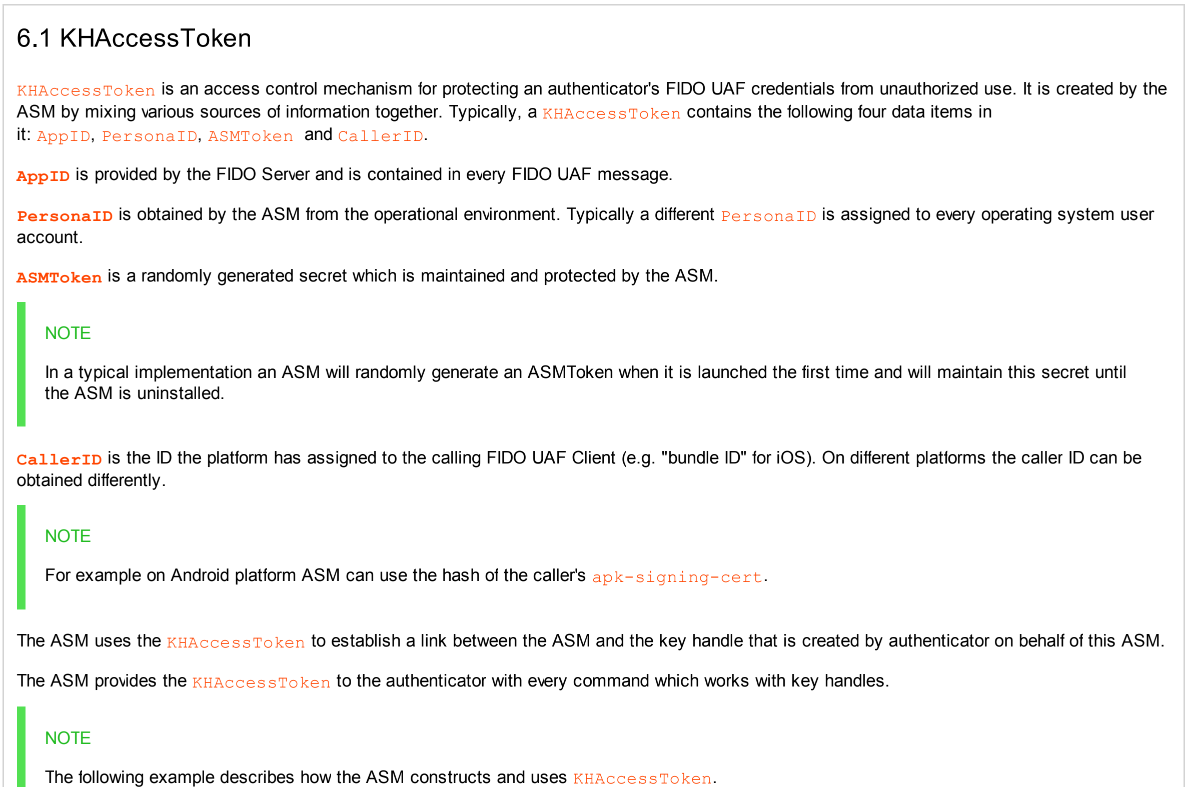
This technique allows making sure that registered keys are only accessible by the caller that originally registered them. A malicious App on a mobile platform won't be able to access keys by bypassing the related ASM (assuming that this ASM originally registered these keys).

The KHAccessToken is typically specific to the AppID, PersonaID, ASMToken and the CallerID. See [UAFASM] for more details.

NOTE

On some platforms, the ASM additionally might need special permissions in order to communicate with the FIDO Authenticator. Some platforms do not provide means to reliably enforce access control among applications.

【ASM API】


6.1 KHAccessToken

`KHAccessToken` is an access control mechanism for protecting an authenticator's FIDO UAF credentials from unauthorized use. It is created by the ASM by mixing various sources of information together. Typically, a `KHAccessToken` contains the following four data items in it: `AppID`, `PersonaID`, `ASMToken` and `CallerID`.

`AppID` is provided by the FIDO Server and is contained in every FIDO UAF message.

`PersonaID` is obtained by the ASM from the operational environment. Typically a different `PersonaID` is assigned to every operating system user account.

`ASMToken` is a randomly generated secret which is maintained and protected by the ASM.

NOTE

In a typical implementation an ASM will randomly generate an ASMToken when it is launched the first time and will maintain this secret until the ASM is uninstalled.

`CallerID` is the ID the platform has assigned to the calling FIDO UAF Client (e.g. "bundle ID" for iOS). On different platforms the caller ID can be obtained differently.

NOTE

For example on Android platform ASM can use the hash of the caller's `apk-signing-cert`.

The ASM uses the `KHAccessToken` to establish a link between the ASM and the key handle that is created by authenticator on behalf of this ASM.

The ASM provides the `KHAccessToken` to the authenticator with every command which works with key handles.

NOTE

The following example describes how the ASM constructs and uses `KHAccessToken`.

- During a Register request
  - Append AppID
    - KHAccessToken = AppID
  - If a bound authenticator, append ASMToken, PersonaID and CallerID
    - KHAccessToken |= ASMToken | PersonaID | CallerID
  - Hash KHAccessToken
    - Hash KHAccessToken using the authenticator's hashing algorithm. The reason of using authenticator specific hash function is to make sure of interoperability(互用性) between ASMs. If interoperability is not required, an ASM can use any other secure hash function it wants.
    - KHAccessToken=hash(KHAccessToken)
  - Provide KHAccessToken to the authenticator
  - The authenticator puts the KHAccessToken into RawKeyHandle (see [UAFAuthnrCommands] for more details)
- During other commands which require KHAccessToken as input argument
  - The ASM computes KHAccessToken the same way as during the Register request and provides it to the authenticator along with other arguments.
  - The authenticator unwraps the provided key handle(s) and proceeds with the command only if RawKeyHandle.KHAccessToken is equal to the provided KHAccessToken.

Bound authenticators MUST support a mechanism for binding generated key handles to ASMs. The binding mechanism MUST have at least the same security characteristics as mechanism for protcting KHAccessToken described above. As a consequence it is RECOMMENDED to securely derive KHAccessToken from AppID,ASMToken, PersonaID and the CallerID.

NOTE

It is recommended for roaming authenticators that the KHAccessToken contains only the AppID since otherwise users won't be able to use them on different machines (PersonaID, ASMToken and CallerID are platform specific). If the authenticator vendor decides to do that in order to address a specific use case, however, it is allowed.

Including PersonaID in the KHAccessToken is optional for all types of authenticators. However an authenticator designed for multi-user systems will likely have to support it.

上文中的Bound Authenticator应该是相对于Roaming Authenticator来说的？

RawKeyHandle:

6.1 RawKeyHandle

RawKeyHandle is a structure generated and parsed by the authenticator. Authenticators may define RawKeyHandle in different ways and the internal structure is relevant only to the specific authenticator implementation.

RawKeyHandle for a typical **first-factor bound authenticator** has the following structure.

Depends on hashing algorithm (e.g. 32 bytes)	Depends on key type. (e.g. 32 bytes)	Username Size (1 byte)
KHAccessToken	UAuth.priv	Size

Table 5.1: RawKeyHandle Structure

First Factor authenticators MUST store Username inside RawKeyHandle and Second Factor authenticators MUST NOT store it. The ability to support Username is a key difference between first-, and second-factor authenticators.

RawKeyHandle MUST be cryptographically wrapped before leaving the authenticator boundary since it contains the user authentication private key (UAuth.priv).