

# DigitalPersona Application Note:

## One Touch I.D. Duplicate Enrollment Check

### Before You Begin

Before reading this document, we suggest that you familiarize yourself with the One Touch for Windows API and the One Touch I.D. API as well as their respective basic terms. A good way to do this, is by reading the SDK product documentation (included with your product) listed below. Depending on your development environment, you will only need to review one of the Developer Guides for each of the two products:

#### *One Touch for Windows*

- [Fingerprint Guide.pdf](#)
- [One Touch for Windows SDK .NET Developer Guide.pdf](#)
- [One Touch for Windows SDK C-C++ Developer Guide.pdf](#)
- [One Touch for Windows SDK COM-ActiveX Developer Guide.pdf](#)
- [One Touch for Windows SDK Java Developer Guide.pdf](#)

#### *One Touch I.D.*

- [One Touch for Windows SDK .NET Developer Guide.pdf](#)
- [One Touch ID SDK Developer Guide.pdf](#)

### DEC Enrollment Check Overview

The “Duplicate Enrollment Check” (DEC) is a feature exposed by the One Touch I.D. 2.1.1 API to allow businesses to reduce the potential of fraud. It allows the application to compare fingerprints being registered to fingerprints of previously registered users.

The actual duplicate enrollment check functionality is exposed by the One Touch I.D. SDK’s overloaded “Identify()” method.

As viewed from the Visual Studio Object Browser:

```
public DPFP.ID.CandidateIDList Identify(DPFP.FeatureSet FeatureSet)  
Member of DPFP.ID.Identification
```

```
public DPFP.ID.CandidateIDList Identify(DPFP.Template Template)  
Member of DPFP.ID.Identification
```

### DEC Pre-Enrollment (1:Many Supervised)

This type of DEC is well suited for a high risk environment. Enrollment is supervised and the workstation is visible strictly to the supervisor. In this setup, the DEC is used primarily for fraud prevention, where users try to register under multiple accounts to gain immediate access to sensitive information and

resources. The DEC can also be used in less “strict” environments, to reduce the likelihood of false accepts, by encouraging users to enroll fingers that do not match other registered prints.

#### Example Scenario:

An application has the task of registering and issuing driver’s licenses for first time driver applicants. This type of application can be the target of fraud since there is potential incentive to provide driver’s licenses to non-qualified individuals. This typically occurs when one individual takes the license tests for other would be drivers. Such an application cannot risk providing a license to the incorrect person as this can result in increased highway fatality rates. Therefore a duplicate enrollment check should be performed immediately at registration; requiring all users to register their prints (either a pre-determined subset or all 10) is key to an accurate DEC.

Workflow of a DEC for a driver’s license issuing system:

1. Application must be supervised.
2. Enrollee will provide identity information.
3. If the information is valid, continue on; else reject enrollee.
4. Supervisor and application instruct the user to press 4 pre-determined fingers to the sensors. The fingers must be presented one at a time, as the reader cannot capture multiple prints in a single press.
5. Application captures the feature sets and sends the DEC request to the server.
6. The server calls “Identify(featureSet)” four times to validate the prints do not already exist in the database.
7. If all 4 prints pass the DEC, have the enrollee register one of the 4 predetermined fingers (preferably the right middle finger). At this point, authentication is complete and there is no need to continue with the next steps.
8. If the fraud check fails, the supervisor is to perform a more stringent examination of enrollee’s identity. If the fraud check fails more often than it should, the system administrator should be able to adjust the default false accept rate (FAR) for the DEC.
9. If the supervisor determines the enrollee’s identity is legitimate and doesn’t exist in the user database, the enrollee is to register one of his\her fingers that do not match any fingers in the database. This is done by simply sending the template to the server for another DEC check. The supervisor and application should inform the enrollee to only use the registered finger for identification purposes.

Code example: (The code contained in this document is to be treated as pseudo-code).

```
//Handles DEC request against an acquired featureset
public bool _RequeFtrsDEC(byte[] ftrBytes)
{

    DPFP.FeatureSet ftrSet = new DPFP.FeatureSet();
    ftrSet.DeSerialize(ftrBytes);
    DPFP.ID.CandidateIDList candidateList = null;

    candidateList=idSet.Identify(ftrSet);
    if (candidateList.Count > 0)
    {
        //Write to server log
        LogDECEnter("DEC Failed\r\n");

        //Compose response, F=Fail
        Response.Write("F");
    }
}
```

```

        foreach(DPFP.ID.UserID candidateID in candidateList)
        {
            //LogDECEnter is a function you should define for logging purposes
            LogDECEnter(candidateID.ToString() + GetDate() + "\r\n");
            Response.Write(candidateID.ToString());
        }
    Else
        //Compose response, P=Pass
        Response.Write("P/0");
}

//Handles DEC request against a registration template
public bool _RequestTemplateDEC(byte[] templateBytes)
{
    DPFP.Template template = new DPFP.Template();
    template.DeSerialize(templateBytes);
    DPFP.ID.CandidateIDList candidateList = null;

    candidateList=idSet.Identify(template);
    if (candidateList.Count > 0)
    {
        //Write to server log
        LogDECEnter("DEC Failed\r\n");

        //Compose response, F=Fail and log the user(s) the print matched against
        Response.Write("F");
        foreach(DPFP.ID.UserID candidateID in candidateList)
        {
            LogDECEnter(candidateID.ToString() + GetDate() + "\r\n");
            Response.Write(candidateID.ToString());
        }
    }
    Else
        //Compose response, P=Pass
        Response.Write("P/0");
}

```

## DEC Pre-Enrollment (1:Many Non-Supervised)

This approach is for those applications where it is impossible to have supervised enrollment. The typical motive for such applications to use One Touch ID is to allow users to login without having to enter their username and password – and to prevent stealing of account credentials (either via key-logger software or brute force attacks). Applications that fall into this usage scenario are usually web based. Any system that is “public” not only benefits from the convenience of biometrics, but also guarantees the millions of individuals that don’t have a DigitalPersona fingerprint reader will have no way of hacking into your customer accounts (assuming the application has a ‘Fingerprint Only’ or similar policy).

Workflow requirements of a DEC for a web forum with sensitive information:

1. Allow users to register at least two fingers. Suggest that they use middle or index fingers.
2. Upon creation of the registration fingerprint template, send a DEC request to the server.
3. If both registration fingerprint templates pass the DEC, save those templates to the user’s database record.
4. If the registration fingerprint templates fail the DEC, have the user register other fingers until two registration templates that pass the DEC are created. Save the registration templates to the user’s database record and inform the user to only use those particular

fingers that were successfully registered.

Note: The user may realize he can use the fingers that failed the DEC to login as another user. Please see the section “Reinforcing Security” at the bottom of this document for more information.

### DEC Post Enrollment (Many:Many)

This approach is well suited for less strict environments where user convenience is the top priority and for those customers who have pre-existing databases of registered fingerprints wherein they need to find already existing duplicates.

Workflow requirements:

1. A separate workstation (not the authentication server itself) should be used to perform a M:M DEC. The workstation must meet minimum hardware requirements (See “Additional Memory Requirements” of the One Touch ID SDK documentation.)
2. An executable or service should be scheduled at the workstation to perform the DEC at periodic intervals (nightly, weekly, or monthly.)
3. Executable should load user registration templates into memory and perform a sequential DEC for every user template against all other user templates at a strict FAR. This will return those users who may have potentially registered under multiple accounts.

Actual code may look something like (assuming the user collection and identification list are already filled):

```
DPFP.ID.User user=null;
DPFP.ID.CandidateIDList candidateList = null;

foreach (DataRow row in tblUsers.Rows)
{
    user = userCollection[row["UserID"]];

    //Call Identify() against each user's template(s)
    foreach (DPFP.ID.FingerPosition finger in user.FingerPositions)
    {
        candidateList = idSet.Identify(user.Template(finger));

        //Write data to file or table
        foreach(DPFP.ID.UserID candidateID in candidateList)
        {
            //WriteDECEntry is a function you would define
            WriteDECEntry(user.ID, candidateID.ToString());
        }
    }
}
```

### Reinforcing Security

Though this document is not intended to address identification, the DEC can play an important role in improving identification accuracy. Based on DEC results, your application can become more “adaptive” when it comes to handling security threats. Ultimately what you decide will be based on customer needs and development resources.

Adaptation is primarily achieved through a second pass of authentication. This is performed by calling the `DPFP.Verification.Verify()` method of the One Touch for Windows SDK after your application calls the `DPFP.ID.Identification.Identify()` method. By using such a mechanism an administrator can have control over recognition policies, allowing for a highly flexible system.

Example Policies:

1. Administrator cannot adjust the FAR. The FAR can only be adjusted by support personnel at your company.
2. Administrator can set a base identification FAR for all users, no per user policy (easiest to implement).
3. Administrator can set a base FAR for all users, in addition to modifying individual user FARs.
4. Administrator can set a base FAR for all users, and allow users to lower (lower=higher security) their own FAR. Obviously if users are given this flexibility, you will need to write your application in a way as to prevent users from locking themselves out. Consider the strength of exposing an “auto-adjust” and “default” option as opposed to letting users adjust their FAR directly. This implementation is more directed towards large identification sets and distributed/web based applications in which it is impractical for an administrator or support technician to adjust individual user accounts.

## How to Adjust the FAR

One Touch for Windows SDK:

```
static int PROBABILITY_ONE = 0x7FFFFFFF;  
DPFP.Verification.Verification verification = new  
DPFP.Verification.Verification();  
  
//Set the FAR to 1/1,000,000 (The OTW default is 1/100,000)  
verification.FARRequested = PROBABILITY_ONE / 1000000;
```

One Touch I.D. SDK:

```
DPFP.ID.Identification IDList = null;  
  
//Set the FAR to 1/25,000 (The OTID default is 1/10,000)  
IDList = new DPFP.ID.Identification(ref userCollection);  
IDList.FalseAcceptProbability=PROBABILITY_ONE/25000;
```

### IMPORTANT NOTE\*\*

Because applications and environments vary, we strongly suggest fine tuning your overall identification and DEC process to find what best fits your customers. Incorrectly modifying the false-accept rate (FAR) can result in a high false accept rate or a high false reject rate, yielding reduced security and a poor user experience.

*For further assistance please post any questions you may have to the Developer Connection web forum.*

<http://www.digitalpersona.com/webforums/index.php>