Global Information Assurance Certification Paper

Build Non-Repudiable Application Audit Logs

Thomas Zhang

Friday, February 27, 2004

GSEC Practical
Version 1.4b – Option 2 (Case Study)

Summary

A vendor developed application, the Mugshot System, used by a large metropolitan police department was evaluated for its security implementations. It was found that the audit log function only logs the intent of a user rather than the actual records viewed by a user. This practice is not acceptable for a typical police application since the audit logs may become evidence presented in the court of law.

This paper documents the shortfalls of the audit log function of the Mugshot application, provides a solution for a true audit trail, and illustrates the audit results logged by the new audit log functions.

Introduction

CIA (Confidentiality, Integrity, and Availability) are the three important properties or requirements of information security [1]. Applying these three requirements to a software application, confidentiality requires that the application and functions within the application can only be accessed by authorized users (authentication and authorization); integrity requires the information be retained to its original accuracy; and availability requires the application can be timely and reliably accessed by the authorized users.

In addition to CIA, a software application handles business transactions. Non-repudiation of the transactions is essential to the business. Authentication and authorization in an application play an important role to safe-guard the confidentiality; while the audit trail should provide the non-repudiable evidence. The audit trail can also provide the trail of records for investigation of misuse.

In a law enforcement business, the audit trail becomes even more important because public safety is at stake.

As part of on-going effort of securing the information systems for this police department, the author was tasked with evaluating the security implementation of all law enforcement applications used in this department. While several applications were evaluated, this paper concentrates on the newly implemented application, Mugshot application, for its audit log function.

To provide a background to the reader, the Mugshot application is briefly described below.

Application Description

The Mugshot System was implemented in February, 2003. The general perceptions and feedbacks from the police personnel are very positive. The major functionalities are outlined below:

- Capture. This is the main function of the application which allows lockup-keeper to capture the arrestee's photos which include front, side and up to four additional photos for scar/mark/tattoos. The arrestees are queued for

each police lockup. If an arrestee is not available for photographing for some reason, a sergeant is called to put this specific arrestee into bypass status.

- Bypass. This function is available to police desk sergeant to put an arrestee into bypass status or remove an arrestee from bypass status.
- Sex Offender Registration (SOR). By law, all the convicted sex offenders must report to their local police department to register themselves at least once a year and whenever they move. The SOR function allows the police personnel to take photos of the sex offenders when they come in for registration.
- Criminal Search. This function is only available to detectives and other approved law enforcement personnel. It allows the user to search criminal photos based on case booking number, individual identification number (based on fingerprint), age, race, sex, arrest date, and/or personal appearance factor such as height, weight, skin color, etc.
- Lineup. This function is available to detectives for them to put together a virtual lineup of photos. The lineups can be saved and retrieved.
- Photo quality audits. This function is available to commanders or their designees at each district for them to evaluate the photo quality taken by each individual lockup keeper or at each lockup.
- IAD. This function is only available to very few personnel in Internal Affairs Division for them to query the photos of sworn police personnel. If needed, a virtual lineup can also be done through this function.

As seen from the above-outlined functionalities of this application, highly sensitive information is stored and accessed through this application. Maintaining the security and audit trail should be an essential part of the application to protect the public safety.

Problem Discovery

Many law enforcement applications used in this department were evaluated for their security implementation. It was found that the authentication and authorization of most applications were implemented fairly well even though improvements can be made to some of the applications (Mugshot is one of them). However, the audit logs of most applications were done poorly.

To illustrate the problem, the author logged into the Mugshot application and performed searches based on two scenarios.

The first search was for white male arrestees who were born between 01-Jan-1993 and 01-Jan-1994. The search screen is shown by Figure 1. Since the search was limited to Juveniles, few results should be returned if any. As expected, only 6 records returned. Figure 2 shows the search results (most of the fields were erased to protect their identities).

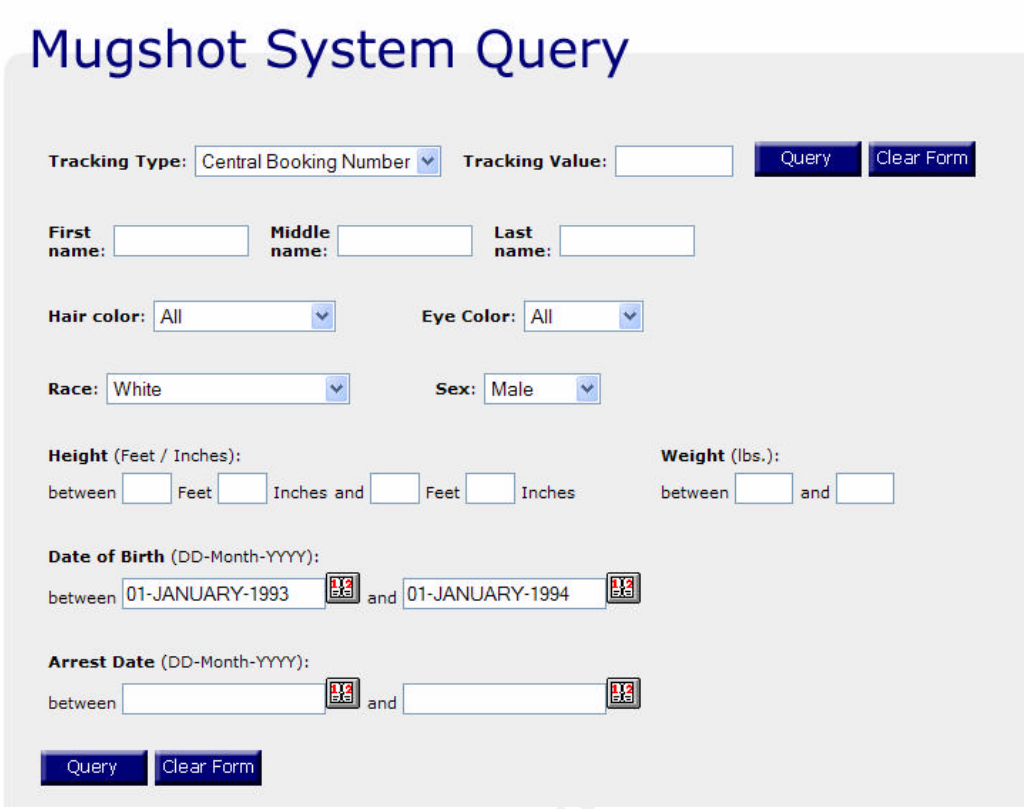


Figure 1. Search Screen shot

Figure 3 shows the audit log entry related to this search. It shows that user "dp01101" performed a search on arrestees at "24-Feb-2004 10:01:21" from IP address "xxx.xxx.xx.xx". The search criteria were

1. Race=whi;
2. Sex=m;
3. DOB_LOW='01-JANUARY-1993'; and
4. DOB_HIGH='01-JANUARY-1994'.

The audit log entry matches exactly what the application user did (or intent of the search). This audit log entry is going to be saved in the database for the future. It will be the same today, tomorrow, or 10 years from now. After all, that is what an audit log is supposed to be. But three important aspects of audit trail were not recorded:

1. **The actual sql statement**. The audit log entry did not record the actual sql statement. It will become difficult (if not impossible) to reconstruct the sql statement in the future. The reader may argue that the actual sql statement can be extracted from the application. But what if the investigation is conducted after the application has gone through several version changes and there was no original code available.
2. **The actual dataset returned.** It did not record how many records were returned by the search, how many records were viewed, and which records were viewed. The number of arrestees who meet the search criteria will increase as the year progresses. If the application user is

investigated for wrong doings on 24-Feb-2004 some time in the future, the audit log entry can only tell the investigator what the application user did, but it cannot tell the investigator what records the user actually *saw*. Without this information, this piece of evidence can be easily disputed and likely be thrown out of court.

3. **The actual dataset viewed.** It did not record the records viewed by the user.

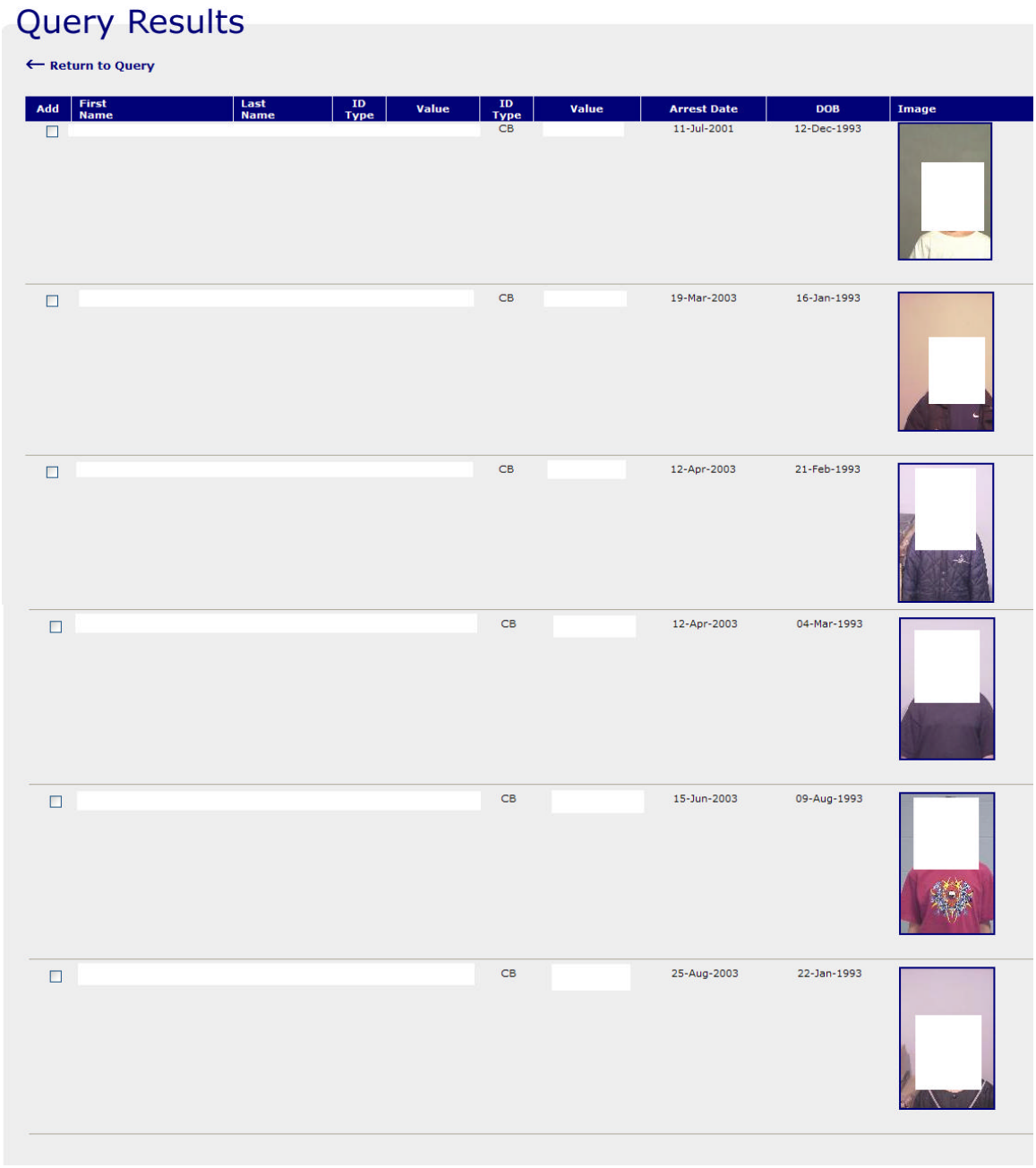


Figure 2. Search Results (scenario I)

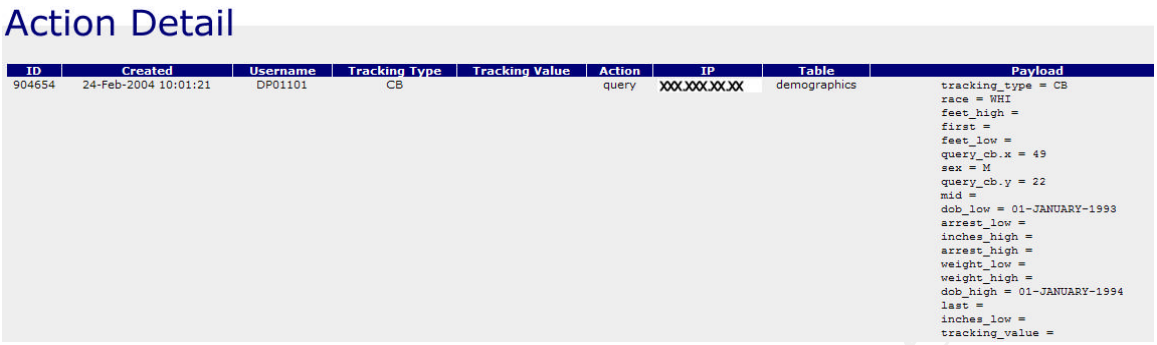


Figure 3. Audit log entry

The second scenario is to search for all male whites who weighed between 140lb and 150lb and arrested between “01-JANUARY-2002” and “10-JANUARY-2002” (Fig. 4). The search results returned 172 hits but only 10 were shown on the first screen (Fig. 5). To see any records beyond the first 10, the link at the bottom of the screen “Next set 11 – 20 of 172” must be clicked. However, in this scenario, the application user never went beyond the first page.

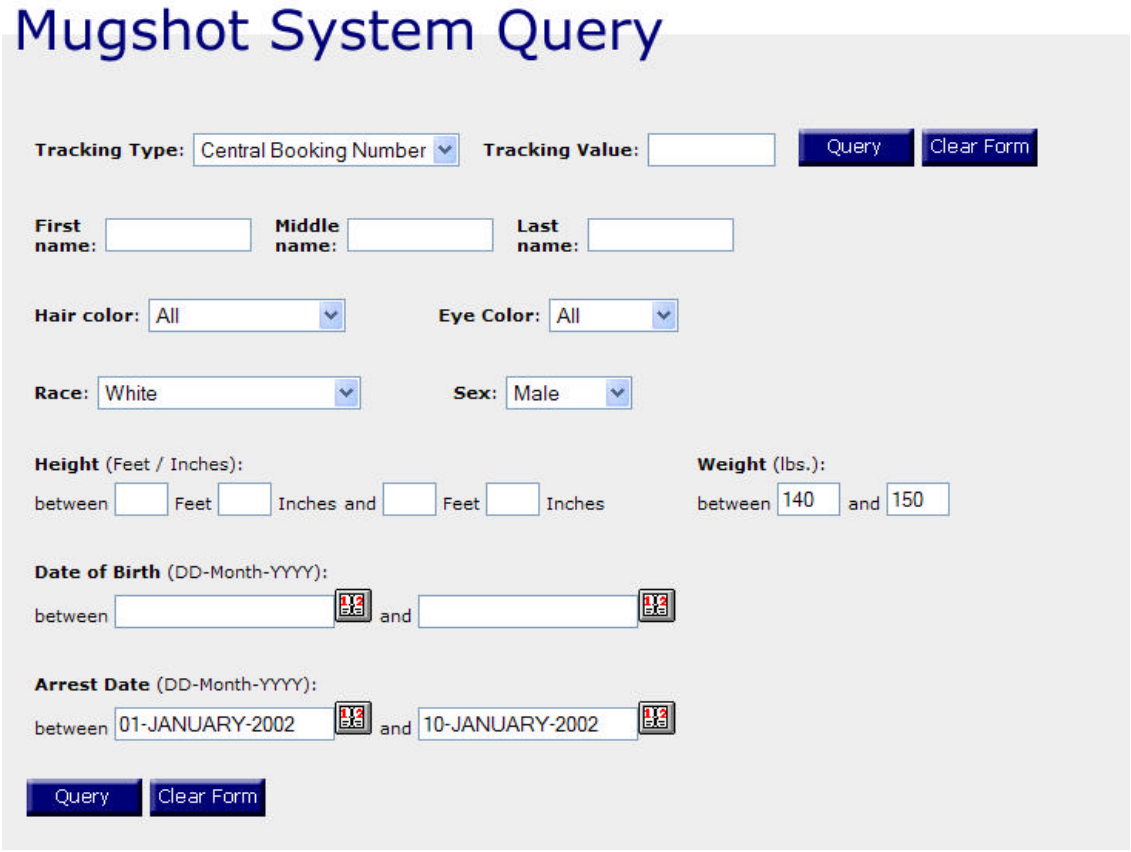


Figure 4. Search based on scenario II

Figure 5. Query results returned (scenario II, 10 of 271 displayed)

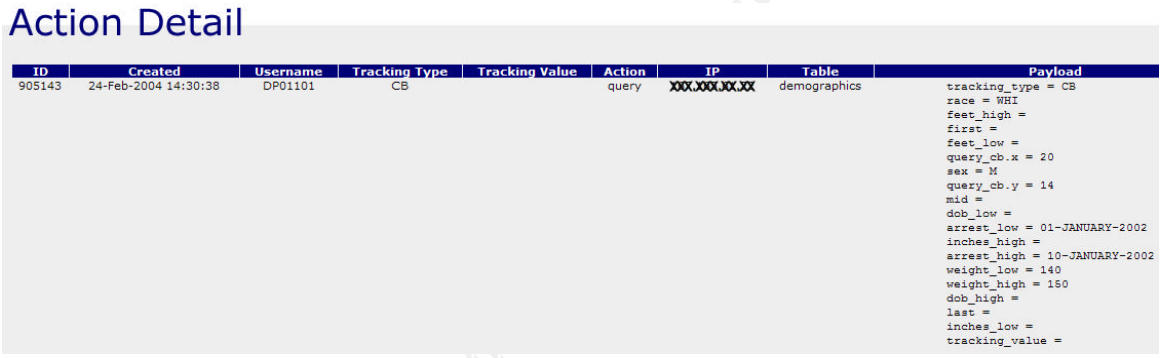


Figure 6. Audit log entry (Scenario II)

Let us take a look at the audit log entry (Fig. 6). The audit log entry (Fig. 6) shows that user “DP01101” did a search at “24-Feb-2004 14:30:38” based on search criteria of

1. race = WHI
2. sex = M
3. arrest_low = 01-JANUARY-2002
4. arrest_high = 10-JANUARY-2002
5. weight_low = 140 and
6. weight_high = 150.

Again, the **actual sql statement**, **returned dataset**, and **viewed dataset** were not logged.

Let us assume that an investigation is initiated one month later. It is required to find out when and who viewed the records of an arrestee. From the audit logs, it was found that the user “DP01101” did a search on 24-Feb-2004. The investigator performed the same search and found the arrestee of interest was on page 5 (101 – 110 of 172). Based on the audit log information, it looks like the application user may have viewed the record of the arrestee of interest as the

audit log entry suggests. However, the user never viewed any record beyond page 1 (1 – 10 of 172) (see Fig. 5).

These scenarios illustrate how the audit log was not enough because the rule of "non-repudiation" was broken.

Solutions Available

To correct the problem, several possible solutions were evaluated. They are listed below.

Vendor Re-program

The initial reaction was for the security team to ask the vendor to re-program the audit section of the application. However, it was found that the vendor was out of business and the original programmer had joined the air force.

Oracle 8i Audit Function

Oracle8*i* database was used as the backend database of the application. Oracle database has audit functionality available. Oracle audit function enables the database administrator (DBA) or database security administrator (DSA) to audit the database activities such as "insert", "update", "delete", "select", and many other activities.

Oracle documentation **Oracle8*i* Administrator's Guide** section 24 "auditing database use" [2] outlined the use of Oracle audit. For example, to audit all "select" statements on the emp table in Scott's schema, following commands were issued (results were formatted for clarity):

```
SQL> connect / as sysdba;
SQL> audit select on scott.emp by access whenever successful;
SQL> conn scott/tiger;
SQL> select * from emp;
EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
----- ---------- --------- ---------- --------- ---------- ---------- ----------
 7369 SMITH      CLERK           7902 17-DEC-80        800                    20
 7499 ALLEN      SALESMAN        7698 20-FEB-81       1600        300         30
 7521 WARD       SALESMAN        7698 22-FEB-81       1250        500         30
3 rows selected.
SQL> connect / as sysdba;
SQL> select os_username,username,terminal, timestamp,owner,
  2  obj_name,action_name,sessionid,entryid,statementid,
  3  returncode from dba_audit_Object;

Oracle SCOTT pts/2 23-FEB-04 SCOTT EMP SELECT 355 1 7 0
```

The audit example above tells us that a user logon to the OS as "oracle" on terminal pts/2 (remote login session) and to the database as user "scott" issued a "select" statement on table "emp" on 23-Feb-2004 with a return code of "0" (For Oracle, 0 means successful). The session id was 355, entry id was 1 (first audit log record) and statement id was 7.

The audit functions provided by Oracle were relatively easily to perform and the performance impact was relatively minor. It is also highly flexible. The DBA or DSA can pick and choose the database objects and actions to audit. However, one of the critical pieces of information, how many records were returned, was not collected or stored.

The Oracle audit functions provide very accurate results in terms of user information for a client-server application. In a multi-tiered application (for example, web-based), "shared schema" is often used. In such cases, the application user is not the same as the database user; the audit information is virtually useless. In reference [2], Oracle tried to address the issue and offered the audit option like:

```
AUDIT SELECT scott.emp BY appserve ON BEHALF OF jackson;
```

Where *appserve* is the middle tier application server and *jackson* is the user we want to audit. However, for applications like the Mugshot application which has more than 15,000 users, to issue this kind of statement for all current users and future users would be a management nightmare.

Oracle Fine-Grained Audit (FGA)

Oracle fine-grained audit (FGA) is an Oracle9i new feature which provides the DBA or DSA a tool to audit "select" statements with great detail. The DBA or DSA has fine-grained control over which table to audit for what kind of actions. A 3-part series by *Arup Nanda* [3, 4, 5] provide an excellent introduction of FGA including its uses and improvement in the latest Oracle database release Oracle 10g. FGA records the statement, system change number (SCN number), Transaction ID and timestamps. This information can be used as the input to the Oracle flashback query and LogMiner to find the actual datasets.

It seemed that our three requirements (logging the actual sql statement, the search result dataset, and viewed dataset) were met. Two caveats eliminated this solution for the Mugshot application:

1. FGA is only available for Oracle9i and up; and
2. FGA relies on LogMiner to reconstruct the dataset. It effectively requires that Oracle archive logs are kept forever which could be cost prohibitive and a management nightmare.

For multi-tiered applications, FGA allows the application to set the client username into the database session records which in turn can be used to identify the true username [4]. This may partially solve the "shared schema" issue associated with multi-tier applications. However, for applications that not only share the "schema" but also share the database connections, it still has a problem to identify the true application user.

Oracle Selective Audit (SA)

Selective audit was developed by Oracle consulting as part of the offerings to their customers. The security team was given a copy of this software for evaluation. Oracle SA combines the Oracle audit function, FGA, Flashback query, and LogMiner together and puts a GUI wrapper around these tools. Similar to FGA, Oracle selective audit logs the actual Oracle statements rather than the statement type as Oracle audit feature does.

The new Oracle9i feature, flashback query, allows the database user to query the database at a point in time in the past [6] for a limited time. This time limit could range from several minutes to several hours or days depending on how the database is configured. The longer the time, the more space is needed to keep the history.

Oracle LogMiner (available on Oracle8i) utilizes the fact that Oracle saves the transaction information into redo logs. Most Oracle database administrators archive the redo logs for database recovery purposes. Oracle LogMiner feature makes use of the transactional information saved in the redo logs. It allows an Oracle DBA or privileged users to mine the transaction information, to reconstruct the data to any time in the past as long as the archived redo log files are available (there are some limitations) [6].

Built based on these two Oracle features (flashback query and LogMiner) and the actual sql statements it logged, Oracle selective audit can reconstruct the results returned by a specific Oracle query statement. This meets the Non-Repudiation requirements. However, three important issues prevented this solution from being implemented:

1. Flashback query on which the Oracle selective audit was built based is only available for Oracle9i and newer. The Mugshot System uses Oracle8i as its backend database. There is no schedule to migrate it to Oracle9i.
2. The problem associated with “shared schema” is still not resolved fully.
3. Oracle SA relies on archived redo logs to get the information about the dataset return or viewed. This effectively requires the Oracle redo log being kept forever. For an application which generates 10GB of redo logs a day, it would be a management nightmare and cost prohibitive to keep all the archived logs forever.

In-house Development

The Mugshot application was developed using open source (Apache with Mod_perl). All the Mugshot application source code is available. It is possible for the development to be done in-house.

After evaluation of the above available solutions, in-house development was chosen. One added benefit of this approach is that this solution can be applied to other applications utilized in the department.

Approaches

The goals of this project were very simple:

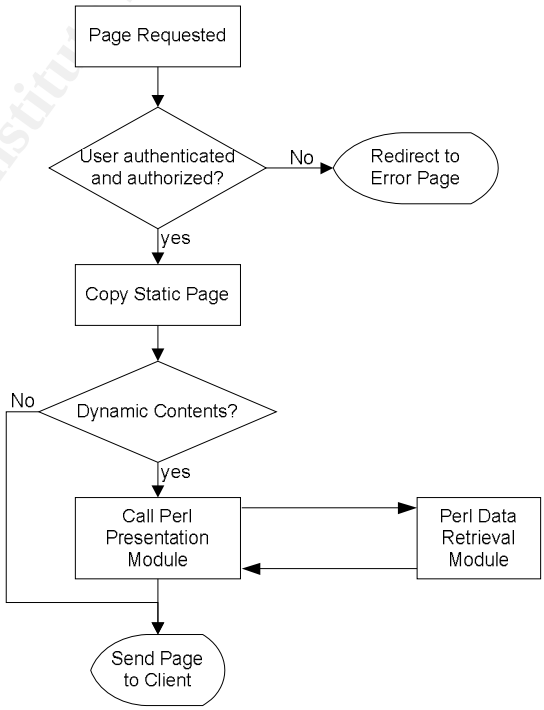
- Program a new audit module which records 1). The actual sql statements; 2). The dataset returned, and 3). The dataset viewed.
- Find all function calls to the existing audit module and replace it with or add a new function call to the new audit module.
- Program a new interface to mine the audit log entries.

Since this application was developed by an outside vendor. The following procedures were taken:

- Understand the application code and the database objects, especially the code snippets related to the audit logs.
- Build a development environment.
- Program and test the solutions
- Publish the changes and verify the results

Understanding the Existing Program

The Mugshot application was written in perl and mod_perl. All the program logic was written in a format of perl library. The application presentation was in a format of shtml which contains static html contents and dynamic html contents much like asp or jsp. When the Apache mod_perl server processes a web page request, it first calls the perl authentication and authorization module. If passed, the Apache mod_perl server continues processing the shtml page. The static contents get copied. For dynamic contents, Apache mod_perl server calls appropriate perl modules which in turn generate the dynamic contents.



There are two sets of perl libraries: 1). the data retrieval library which is responsible for fetching, inserting, and updating records, user authentication and authorization, etc.; and 2). the presentation library which is responsible for calling various functions and subroutines of the data retrieval library, processing the data, and generating the dynamic contents. The program flowchart is illustrated above.

Now that we understand the general structure of Mugshot application program methodology, let us look at the code snippets related to audit logs.

Following are the related code snippets:

Listing 1:

```
    if ( $criteria_check > 0 ) {
      Sentinel::Defender->event(
                'query',
                time(),
                $pxt->get_remote_host(),
                $user->id,
                'demographics',
                $outbound{tracking_type},
                $outbound{tracking_value},
                \%outbound
    );
```

Listing 2:

```
      sub event {
        my $class = shift;

        my $type = shift;
        my $time = shift;
        my $ip = shift;
        my $uid = shift;
        my $table = shift;
        my $tracking = shift;
        my $tracking_value = shift;
        my $payload = shift;

        my $result = Sentinel::DB::Defender->insert($type, $time, $ip,
      $uid, $table, $tracking, $tracking_value, $payload);

        if ( $type eq 'failed_login' ) {
          Sentinel::DB::Defender->failed_login($uid);
        }

        return(0);
      }
```

Listing 3:

```
      sub insert {
        my $class = shift;

        my $type = shift;
```

```
my $time = shift;
my $ip = shift;
my $uid = shift;
my $table = shift;
my $tracking = shift;
my $tracking_value = shift;
my $payload = shift;

my $dbh = Sentinel::DB->connect($Sentinel::Conf::DB);

my $encoded;

if ( $payload ) {
  if ( ref($payload) ) {
    $encoded = encode_base64( freeze( $payload ) );
  } else {
    $encoded = encode_base64( $payload );
  }
} else {
   $encoded = encode_base64( 'none' );
}

my $id = Sentinel::Utils->sequence('defender_seq');

if ( $uid ) {
  my $sql = "insert into defender
          ( id,
          type_id,
          originator_ip,
          user_id,
          table_affected,
          tracking_id,
          tracking_value,
          payload )
          ( select
                $id,
                ty.id,
                \'$ip\',
                $uid,
                \'$table\',
                tr.id,
                \'$tracking_value\',
                \'$encoded\'
          from
                defender_type ty,
                defender_tracking tr
          where
                ty.type = ?
                and upper(tr.tracking_type) = upper(?))";

  my $sth = $dbh->prepare($sql);
  $sth->execute($type, $tracking);
  $sth->finish();
  $dbh->commit();
}

return(0);
```

```
}
```

Listing 1 is the code snippet which exists throughout Perl presentation library. *%outbound* is the hash of all searching criteria entered by a user; Sentinel is the name of data retrieval library; Defender is the Perl audit log package within the data retrieval library – Sentinel; and Event is a subroutine inside the Defender package. Among the passing parameters, 'query' indicates that the action type was search, 'demographics' is the main table name in which this action is performed upon. Other parameters are self explanatory.

Listing 2 shows the code for the "Event" subroutine that was called by the presentation library. It in turn calls an "insert" subroutine in another Perl package "Sentinel::DB::Defender" to insert the audit log into the audit log table (which is stored in the Oracle8i database).

Listing 3 shows the audit log subroutine. Column "payload" in audit log table "defender" is of "clob" data type while the payload parameter passed in is a hash or hash reference. This function first base-64 encodes the payload and then inserts the audit log entry into Oracle8i database table.

When a log entry is requested for audit, the data retrieval function will base_64 decode it first and then return the hash value to the presentation library.

Building a Development Environment

The production box of the Mugshot application has the following characteristics,

- Operating system – Sun Solaris;
- Program environment – Perl 5.6.1, shtml, Expat 1.95 xml;
- Database – Oracle8i;
- Web server – Apache (1.3.20) with Mod_perl (1.26)

All the security patches applied. Because a Sun Solaris box was not available to the author, it was decided that a Linux (RedHat9) box could be used as the development platform. However, RedHat9 came with the latest Perl, Apache, and Mod_perl. It was difficult for the author to guarantee the program to be backward compatible with the production environment of the Mugshot application. The author decided to custom build the program environment and the web server to simulate the production environment.

Due to the constraints of this paper it is assumed that the reader has had prior experience with the Linux/Unix operating system and is familiar with installing the open-source software.

Since Perl 5.8.2 comes with the RedHat9 OS, many other programs may depend on this version of Perl. A second Perl environment had to be installed in a different directory. Even though RPM is a better way to install software, RPM only allows the software to be installed at its default directory. The author had to

install Perl 5.6.1 from the source code. The following sequence of commands install Perl 5.6.1 (a stable release) from the source code:

```
wget ftp://ftp.funet.fi/pub/CPAN/src/perl-5.6.1.tar.gz -o/tmp//perl.tar.gz
cd /tmp
gzip -d /tmp/perl.tar.gz
tar xvf /tmp/perl.tar
cd perl-5.6.1
rm -f config.sh Policy.sh
sh Configure -de
make
make test
make install
```

The readme and INSTALL documents that come with the source tar ball provide the detailed installation instructions. Also, the CPAN web-site maintains a very comprehensive FAQ [8]. Instruction on how to install Perl from the source code can also be found on this web-site as well.

Next, the following Perl packages need to be installed:

- MD5
- CPAN
- Digest::MD5
- MIME::Base64
- URI
- Storable
- Bundle::libnet
- HTML::Parser
- DBI
- Bundle::LWP

These Perl modules can be installed using Perl CPAN utility. The following commands will install all the Perl packages listed above:

```
for x in MD5 CPAN Digest::MD5 MIME::Base64 URI Storable Bundle::libnet HTML::Parser DBI \
        Bundle::LWP
do
  echo "$x"
  /usr/local/bin/perl -MCPAN -e "install '$x'"
done
```

Listing 4 shows the command sequences to install Expat, Apache, and Mod_perl. For detailed instructions on how to install Expat, Apache, and Mod_perl, please refer to [8], [9], and [10].

Listing 4.

```
wget http://prdownloads.sourceforge.net/expat/expat-1.95.1.tar.gz -O/tmp/expat.tar.gz
cd /tmp
gzip -d /tmp/expat.tar.gz
```

```
tar xvf /tmp/expat.tar
cd /tmp/expat-1.95.1
./configure
cd lib
perl -pi'.orig' -e 's/\-Wp\,\-MD\,\.deps\/\$\(\*F\)\.pp//' Makefile
cd ../
make
make install

wget http://archive.apache.org/dist/httpd/old/apache_1.3.20.tar.gz -O/tmp/apache.tar.gz
cd /tmp
gzip -d /tmp/apache.tar.gz
tar xvf /tmp/apache.tar
wget http://perl.apache.org/dist/old/mod_perl-1.26.tar.gz -O/tmp/mod_perl.tar.gz
gzip -d /tmp/mod_perl.tar.gz
tar xvf /tmp/mod_perl.tar
cd mod_perl-1.26
perl Makefile.PL EVERYTHING=1 DO_HTTPD=1
make
make test
make install
```

Now, the following additional Perl packages need to be installed.

- XML::Parser
- DBD::Oracle
- Apache::Request
- Frontier::RPC2
- Imager

The following code snippet will accomplish the task:

```
echo "Please insert the name/password@dbname for oracle..."
read ORACLE_USERID
export ORACLE_USERID

echo "Installing DBD::Oracle, Apache::Request and Frontier::XML-RPC"

for x in XML::Parser DBD::Oracle Apache::Request Frontier::RPC2 Imager
do
  echo "$x"
  perl -MCPAN -e "install '$x'"
done
```

Notice that the Oracle connection information was in the code snippet above. For this to be successful, the following needs to be done first:

- Oracle client software must be installed and configured on this box
- A test schema with all database objects (tables, indexes, etc.) must be created on a database server.

Normally, this work is done by an Oracle DBA. For this project, the author installed the Oracle database server on a RedHat9 box. By referencing to [12,

13], the author was able to install Oracle9i database server on RedHat9 with minimal difficulty.

Programming and Testing the Solutions

The goals for the audit logs are clear. All the following activities should be logged:

- All transactions
- All search activities
- Actual records viewed

An additional goal is that this solution may be applied to all possible applications currently used in this department. For this reason, it was decided that a new Oracle schema be created and the audit trail be written to this new schema. This schema can be available for other applications as well. The benefits of this approach are:

- Easy to manage. The data in the audit trail can be independently backed up.
- Easy to protect the audit trail. Only insert privilege needs to be given for applications to log the audit trail. Members of the security group have only "select" privilege. Nobody has "delete" or "update" privileges on the audit log tables.
- Centralize the audit logs for easy audit. The audit logs can be centralized to one database. The audit log functions of all applications can be stored in this centralized database.

When the new audit function was designed, it was assumed that every record can be uniquely identified by a key column or a combination of key columns (primary key). Analysis of all applications used in this police department shows the records displayed to the users can be uniquely identified by one key column of a main database table. It can be determined by the application developers to decide which column or columns are the unique identifier.

Based on the above assumption, the database schema was designed. The following scripts create the database tables for audit log schema:

```
create table app_audit_obj (objid number, appid number, objschema varchar2(30),
objname varchar2(200))
/

create table app_audit_rec(session_id varchar2(50), objname varchar2(200), stmtid
number, keyname varchar2(30), keyvalue clob)
/

create table app_audit_trail (appid number(2),
  session_id varchar2(50),
  timestamp timestamp(6),
  stmtid number,
  app_user varchar2(30),
  db_user varchar2(30),
  os_user varchar2(30),
  hostip varchar2(39),
```

```
  scn number,
  sql_text clob,
  sql_bind varchar2(4000),
  comments varchar2(4000)
)
/
```

For audit logging, only app_audit_rec and app_audit_trail tables are used. The app_audit_obj table is only used as a reference when the logs are audited.

Listing 5

```
sub insert_sql {
  my $class = shift;
  my $pxt = shift;
  my $time = shift;
  my $scn = shift;
  my $key_name = shift;
  my $key_value = shift;
  my $sql_text = shift;
  my $sql_bind = shift;
  my $comments = shift;

  my $dbh = Sentinel::DB->connect($Sentinel::Conf::DB);
  unless($scn) {
    my $dbversion = $Sentinel::Conf::dbversion;
    if ( $dbversion ne '8.1.7' ){
      my $sql_scn = "select DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER
from dual";
      my $sth_scn = $dbh->prepare($sql_scn);
      $sth_scn->execute();
      $scn=$sth_scn->fetchrow();
      #warn "scn scn scn -- $scn";
    } else {
      $scn='';
    }
  }

  my $sessionid = $pxt->session->{__key__};
  #my $appuser = $pxt->{__user__}->{__username__};
  my $appuser = $pxt->user->{__username__};
  my $dbuser=$Sentinel::Conf::username;
  my $osuser='pclogin';
  my $ip = $pxt->get_remote_host();

  my $stmtid = Sentinel::Utils->sequence('stmtid_seq');
  my $sql = "
          insert into app_audit_trail
(appid,session_id,timestamp,stmtid,app_user,db_user,os_user,hostip,
      scn,sql_text,sql_bind,comments)
      values(1,?,to_date(?,'Mon dd hh24:mi:ss yyyy'),?,?,?,?,?,
      ?,?,?,?)
          ";

  my $sth = $dbh->prepare($sql);
```

```
  $sth->execute($sessionid, $time, $stmtid, $appuser, $dbuser, $osuser, $ip,$scn,
$sql_text, $sql_bind, $comments);
  $sth->finish();
  $sql = "
            insert into app_audit_rec (session_id,stmtid,keyname,keyvalue)
            values (?,?,?,?)
        ";
  $sth = $dbh->prepare($sql);
  $sth->execute($sessionid, $stmtid, $key_name, $key_value);
  $sth->finish();
  $dbh->commit();

  return(0);
}
```

Listing 5 shows the function that gets called (see Listing 6 below) to insert the audit logs into the audit trail.

Recall that the audit function is called from the Perl presentation library. The difference is that the old audit function records the user attempts. It logs the user's attempt to perform a function with certain criteria. For example, for a search function, it logs that a user attempted to search using some search criteria. It does not have any record of what the user actually accomplished (such as number of search hits and number of records viewed). The new audit function records the search results that the user actually viewed. The calling block can be originated from the Perl presentation library, but it is much easier if the function call was originated from the data retrieval library.

Listing 6

```
Sentinel::DB::Defender->insert_sql($pxt, $now, $scn, $key_name, $key_value,
$sql, $sql_bind, "fetch multiple images");
```

Each time a web page is displayed (via a mouse click) the function to record the audit information is called. Listing 6 shows the actual calling code snippet, where $pxt is the Apache objects which contains all session related parameters; $now is the timestamps which is assigned by

```
my $now= substr($localtime(),4);
```

and $scn is the Oracle database system change number. Oracle uses scn to identified the transactions. In the calling block of the program, $scn is assigned by:

```
  my $scn;
  my $dbversion = $Sentinel::Conf::dbversion;
  if ( $dbversion ne '8.1.7' ){
    my $sql_scn = "select DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER from
dual";
    my $sth_scn = $dbh->prepare($sql_scn);
    $sth_scn->execute();
```

```
  $scn=$sth_scn->fetchrow();
} else {
  $scn='';
}
```

Where DBMS_FLASHBACK is an Oracle supplied package which can be used to fetch the scn number from the database. Because DBMS_FLASHBACK package is only available for Oracle9i and now Oracle 10g, the program assigns $scn null value if the Oracle version is 8.1.7. Obviously if you have lower version of Oracle, the piece of code needs to be changed. $key_name is the name of the column or columns which can uniquely identify the record. For example, column 'CB' uniquely identifies a record in table "demographics" while columns 'CB' and 'TYPE_ID' uniquely identify a record in table "cb_image". The corresponding variable assignments are:

```
$key_name='CB';

Or,

$key_name='CB:TYPE_ID';
```

Notice the colon between the column names. $key_value stores the column values for the record. Similarly, for a single column unique identifier, $key_value is assigned as:

```
foreach my $r ( @{$row} ) {
  if ( $r->[2] ) {
    $key_value .= $r->[2] . '!';  #column 2 is the unique identifier
  }
}
```

If the unique identifier is a combination of two columns, $key_value is assigned as:

```
foreach my $r ( @{$row} ) {
  if ( $r->[2] ) {
     $key_value .= $r->[2] . ':' . $r->[3] . '!'; #combination of column 2 and 3 is the
                                                  #unique identifier
  }
}
```

$sql is the Oracle sql statement; $sql_bind stores the bind variable values; and "fetch multiple images" is the comments.

Using a Perl array to store the key name and bind variables is a natural choice (from a Perl developers perspective) and initially, it was coded this way. However, after some testing, it was decided it is better to use concatenation of all elements of the array. After all, the purpose of storing audit logs is to be able to mine and extract useful information via some tools. If the array was used, then base_64 encoding would have to be used to store the array into the database in a clob

datatype column. Searching base_64 encoded data stored in a clob database column is slow and difficult. In the concatenated form, the records can be text indexed and therefore Oracle Regex could be used for audit data mining. The concatenated storage format makes it quick and easy to store and search the data in the Oracle database. This format also makes it quick and easy to return a record set into a Perl data structure and for Perl Regex to be applied.

Results

The new audit log function enables the Mugshot application to completely and accurately log the user activities through the application. The audit results will reflect the actual events that happened.

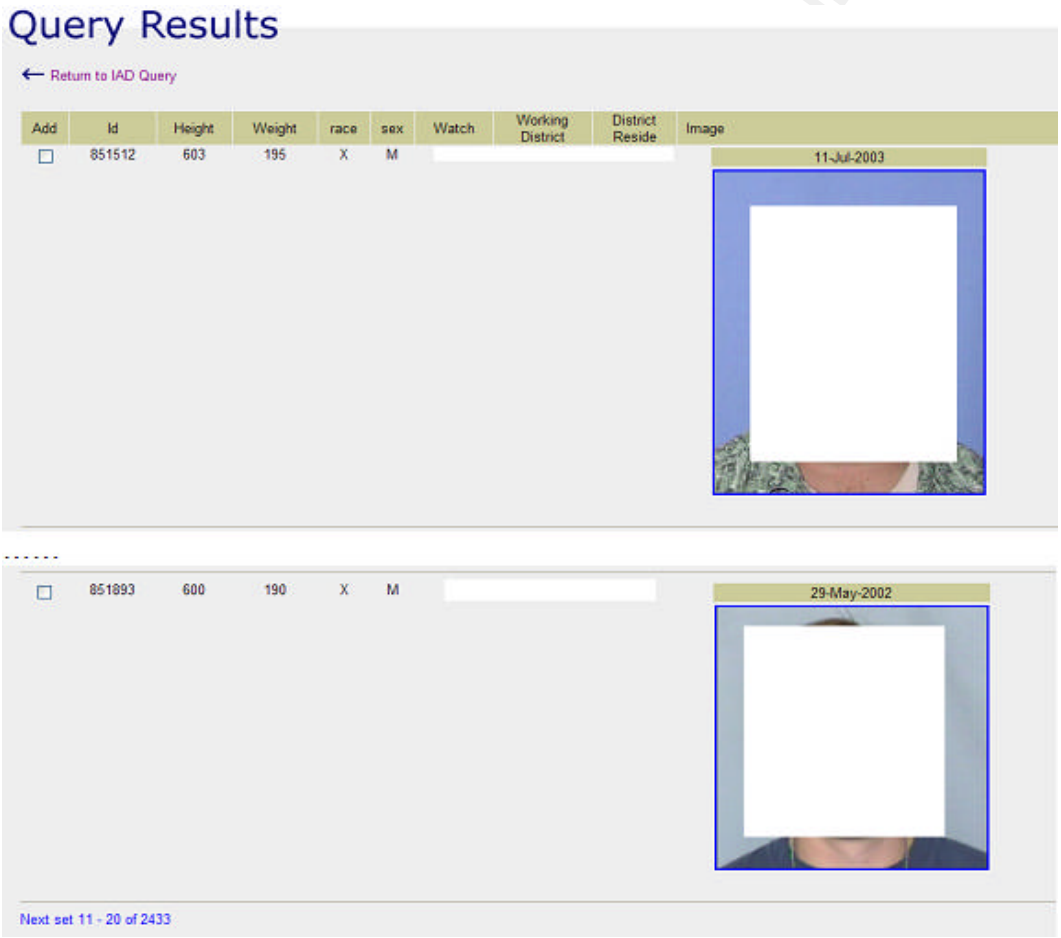


Figure 7 Search result (10 of 2433 displayed)

Figure 8 Audit log entry

The screen shots shown by Figure 7, 8, and 9 show an example of the results of new audit functions. An Mugshot user (logged in as user 'tzhang') performed a search for white male police officers who were born between '01-JAN-1960' and '01-JAN-1970' and worked '02-FEB-2004'. The query returned 2433 records and the first 10 records were displayed to the Mugshot user.

The audit log (Figure 8) shows that user 'tzhang' performed "CR Investigation" type action, 2433 records returned but only 10 records viewed. The last column "Get Detail" is the link which shows the records viewed by the author (Figure 9).

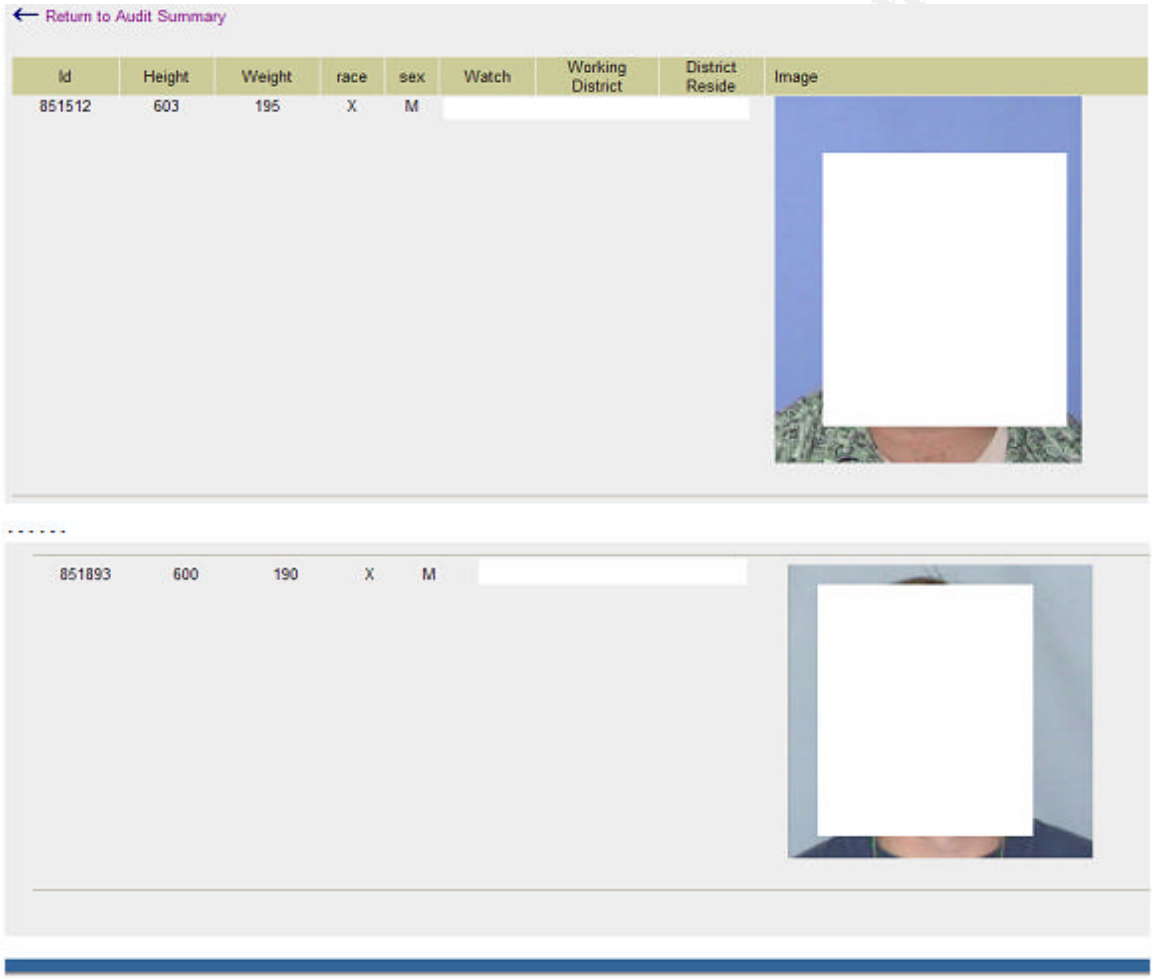


Figure 9 Audit log entry details

There are many other ways to mine the audit logs. The interface shown above is just one of the functions that were programmed for this application. The security team is in the process to:

- Make this audit solution available to other applications so that the audit trails from other applications will be the same as the revised Mugshot application audits.
- Centralize the audit logs. Storing the audit logs in an independent schema allows us to centralize the audit logs.

- Program a comprehensive audit mining application. This audit log mining application will allow a great flexibility of the mining the audit logs. For example, analyzing the access pattern of a particular username, or analyzing a particular record being accessed.

Closing Remarks

Most of the efforts of security professionals are concentrating on the network (LAN or WLAN), hosts on the network (firewall, HIDS, NIDS, router and switches, servers, desktops, etc.). When it comes to application security, we rely on the application developers, software vendors, and software development vendors to implement the security. Depending on the knowledge of the developers, the security implementation of the software used by the enterprise may or may not be desirable or even correct. In the case of the Mugshot application used here in this police department, while it meets most of the functional requirements of the application, it was found the audit trail did not meet our business requirements. Also, there were flaws in the authentication and authorization implementation (corrected as part of this project but not documented here).

While the security professionals still should keep their effort up on securing the corporate networks, a closer look at the security implementation of the applications is warranted, especially for these custom-developed applications.

References

[1] Julian Curmi, Compulsory Cost_Control
http://www.speedyadverts.com/SATopics/html/information_security.html
[2] Oracle8*i* Administrator's Guide http://download-east.oracle.com/docs/cd/A87860_01/doc/server.817/a76956/audit.htm#1136
[3] *Arup Nanda* Fine-Grained Auditing for Real-World Problems
http://otn.oracle.com/oramag/webcolumns/2003/techarticles/nanda_fga.html
[4] *Arup Nanda* Fine-Grained Auditing for Real-World Problems, Part 2
http://otn.oracle.com/pub/articles/nanda_fga_pt2.html
[5] *Arup Nanda* Fine-Grained Auditing for Real-World Problems, Part 3
http://otn.oracle.com/pub/articles/nanda_fga_pt3.html
[6] Oracle9*i* Application Developer's Guide – Fundamental http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96590/adg08sql.htm#10851tals
[7] Using LogMiner to Analyze Redo Logs - Oracle9*i* Database Administrator's Guide http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96521/logminer.htm#24738
[8] CPAN Frequently Asked Questions http://www.cpan.org/misc/cpan-faq.html
[9] Apache Server Frequently Asked Questions
http://httpd.apache.org/docs/misc/FAQ.html
[10] Mod_perl Installation http://perl.apache.org/docs/1.0/guide/install.html
[11] Frequently Asked Questions about Expat
http://www.jclark.com/xml/expatfaq.html

[12] *Roko Roic* Installing Oracle 9iR2 on Red Hat 9
http://www.oreillynet.com/cs/user/print/a/4141
[13] Installing Oracle 9i R2 on Linux http://www.gurulabs.com/oracle-linux.html