LEEDS BECKETT UNIVERSITY

Web History Extraction Artefact Tool

Technical Report

David Michael Keebles

09/04/2015

Table of Contents

1.0 Design

The Web History Extraction Artefact Tool is a GNU/Linux operating system running the Fluxbox front-end GUI. This front-end GUI has been vastly locked down to prevent the user from being able to do anything other than the tools intended purpose. This is to prevent non-technical users from performing unexpected or forensically unsound operations. There are three TTYs running; TTY1 is running the tools background operations, TTY2 is a login terminal to enable technically sound persons to perform advanced operations (it is a full login terminal with root access) and TTY3 is running the Fluxbox GUI.

Mingetty has been installed to allow automatic login as the root user. Fluxbox is automatically started and the tool initiated so no technical knowledge is needed to start the tool. The root user is used as the main user to prevent any need to use the sudoers file; as the tool is heavily locked down and can only perform predetermined operations it would be inefficient to use an ordinary user with the requirement of performing root operations regularly.

The Web History Extraction Artefact Tool has been designed to have two running workspaces; the first is an acquisition workspace which runs a dialog based GUI, the second is a viewing workspace which is running a locked down Firefox that defaults to the PHP servers homepage.

1.1 Acquisition Content

The backbone of the acquisition module within the Web History Extraction Artefact Tool is BASH. This is not very user friendly for non-technical users and as such a GUI in the form of dialog menus, forms, checklists and gauges has been used to ensure the user does not get lost in a sea of options presented in a text based format.

Input from the West Yorkshire Police Hi-Tec Crime Unit resulted in the decision that an output log would be advantageous to the user, however should not contain over-technical information. Basic information such as the found drives and operating systems are displayed here. The tool writes this log to two different places – the main log and a case specific log.

BASH is used as the main scripting language to manage the operations being performed. This is due to its ability to interact with the GNU/Linux system natively. Python is used as the main scripting language to parse the data due to its speed and efficiency.

Originally the tool was designed with the legacy Firefox cache in mind, however after this design Mozilla released a new version of the cache '`cache2`' just after development began. This new cache has been incorporated into the tool at the expense of some Internet Explorer functionality. This decision was made due to the fact that Microsoft are soon releasing a completely rebuffed browser rendering Internet Explorer soon to be obsolete.

The Mozilla Firefox '`cache2`' has recently been turned on by default, and the legacy cache turned off. The legacy cache directory structure has also recently been modified after the research phase. This new structure has also been implemented into the tool.

1.2 Viewing Content

The tool has been designed with two workspaces to allow the examiner to process evidence at the same time as viewing this and any previous evidence at the same time on another workspace. This

design works well, however at some points it overloads the PHP server which resulted in the viewing workspace rendered unusable due to the locked down tool. This required an addition to the PHP design in the form of '`ErrorDocument`' pages being created, this addition to the design allows the examiner to refresh the page once the tool has enough resources to continue.

Originally, the tool was designed to run all as one page with sourced PHP files as necessary.This design is still generally used, however it was found to be efficient to lock down the tools Mozilla Firefox completely to prevent user interaction with anything other than its intended purposes. This has been done using the Mozilla Firefox internal CSS styling and hiding all elements such as the URL bar and the buttons.

After collaborating with West Yorkshire Police Hi-Tec Crime Unit, it was decided that the tool should be designed to give as much information to the user as possible [Refer to section 1.3 for more details on this in regard to HCI theory]. This workspace allows the user to view multiple cases in the form of a navigation bar. Once an operating system is selected, another navigation menu appears which allows the user to view different parsed details such as the web history, Google Analytics cookies and the rebuild cache.

1.3 Implication of HCI Theory

After researching HCI Theory the main text output was designed to be sans-serif based font. For users who may find it difficult to read the font size on unusually sized screen resolutions there have been two main designs implemented, a compact size and a full size. There has also been a large font option implemented.

To make the tool more aesthetically pleasing, some basic images have been used to aid the recognition of different type of page, operating system and browsers.

After collaborating with West Yorkshire Police Hi-Tec Crime Unit, it has been concluded that it is better for the examiner to have as much information displayed as possible. However HCI Theory dictates that this is bad practice, and only up to seven (7) separate pieces of information should be displayed at one time. To mediate between these two, the design of the tool has been focused around a model which separates the pages into sections with each section containing certain types of information.

2.0 Development

While developing the Web History Extraction Artefact Tool three main languages were used in conjunction with each other: BASH, Python and PHP. This section highlights some of the main components of these scripts.

2.1 Acquisition BASH Code

BASH is used as the main scripting language due to its interoperability with the GNU/Linux base operating system.

2.1.1 Forensically Sound Mounting

Before the data on the suspect drives can be accessed, the relevant partitions need to be mounted in a forensically sound read only manner. For GNU/Linux Kernels, FAT32, NTFS, EXT2 etc. file systems

can be mounted with the read only flag to obtain this level of read only protection, however this does not work for EXT[34] file systems if their journals are “dirty”, resulting in them being “fixed” and therefore changed. This BASH script checks that the current file system can be mounted in a forensically sound manner and mounts it using the relevant options.

```
    if [[ ${fs,,} =~ ext[34] ]]; then
        # ext[34] file systems with a dirty journal will be "fixed" on mount; therefore the
loopBack function will not mount in a forensically sound manner. The extFS function prevents this
"feature" from occuring.
        extFS
    elif [[ ${fs,,} == 'fat16' ]] || [[ ${fs,,} == 'fat32' ]] || [[ ${fs,,} == 'ntfs' ]] ||
[[ ${fs,,} == 'ext' ]] || [[ ${fs,,} == 'ext2' ]] || [[ ${fs,,} == 'vfat' ]] || [[ ${fs,,} ==
'msdos' ]] || [[ ${fs,,} == 'reiserfs' ]] || [[ ${fs,,} == 'hfs' ]]; then
        loopBack
    elif  [[ ${fs,,} == 'xfs' ]]; then
        logMsg "  ${partition} contains the unsupported file system XFS which, if it contains a
dirty journal, MAY be impossible to mount in a forensically sound manner - skipping!" 'w' "/tmp/
suspectMountReport"
    elif [[ ${fs,,} =~ 'luks' ]]; then
        logMsg "  ${partition} contains an encrypted file system - skipping!" 'w' "/tmp/
suspectMountReport"
    else
        logMsg "  ${partition} contains an unknown or unsupported file system (${fs}) -
skipping!" 'w' "/tmp/suspectMountReport"
    fi
```

For most file systems, the mount process requires that a loop device to be created from the device. This loop device is then mounted in a read only manner.

```
function loopBack {
    mountPoint
    loop=$(losetup -f)
    # all clean file systems can be mounted (except xfs)
    losetup -r ${loop} ${partition}
    checkError "  An error occurred while attempting to create a loop device for ${partition}!
(UUID: ${UUID})" "" "/tmp/suspectMountReport"
    mount -o ro ${loop} ${mountPoint}
    checkError "  An error occurred while attempting to mount ${partition} in a forensically sound
manner! (UUID: ${UUID})" "  Partition ${partition} has been successfully mounted in a forensically
sound manner. (UUID: ${UUID})" "/tmp/suspectMountReport"
}
```

For EXT3 file systems, a complex command was required to prevent a dirty journal from being fixed involving the backup journal; however with the advent of EXT4, which is backwards compatible to EXT3, this method is now very simple. The ‘`noexec`’ and ‘`noload`’ flags tell mount to ignore any warnings about the journal being dirty and prevents any data whatsoever from being written to the device in these circumstances.

```
function extFS {
    mountPoint
    # ext4 is backward compatible to ext3
    mount -t ext4 -o loop,ro,noexec,noload ${partition} ${mountPoint}
    checkError "  An error occurred while attempting to mount ${partition} in a forensically sound
manner! (UUID: ${UUID})" "  Partition ${partition} has been successfully mounted in a forensically
sound manner. (UUID: ${UUID})" "/tmp/suspectMountReport"
}
```

2.1.2 Obtaining GNU/Linux User Information

While processing GNU/Linux users on a target machine, the ‘`/etc/passwd`’ file was concatenated and each entry was cut to obtain all relevant pieces of information.

```
# get list of user home directories
driveRootPath="/mnt/suspect/${mountPoint}"
linHomeDirsFF="$(cat $driveRootPath/etc/passwd)"
linHomeCount=$(echo "${linHomeDirsFF}" | wc -l)
for ((l=1; l<=$linHomeCount; l++)); do
     ffLinUser=$(echo "$linHomeDirsFF" | head -n${l} | tail -n1)
     linUser=$(echo ${ffLinUser} | cut -d':' -f1)
     linUid=$(echo ${ffLinUser} | cut -d':' -f3)
     linFullName=$(echo ${ffLinUser} | cut -d':' -f5 | cut -d',' -f1)
```

Only users created and/or potentially used by the user for web browsing are relevant to this tools purpose. Therefore all users with a home directory and UID of root (0) or non-system (>=500 and <30'000) are checked for Mozilla Firefox installations.

```
linHome=$(echo ${ffLinUser} | cut -d':' -f6)
if [[ ! $linHome == '' ]]; then
     if [[ $linUid == '0' || $linUid -gt 499 && $linUid -lt 30000 ]]; then
```

Once these users are obtained, their home directories are checked for '`.mozilla`' and, for the latest Firefox version with cache2, '`.cache/mozilla`'. If these directories exist, all profile data is retrieved to be parsed by later scripts.

```
if [[ -d "/mnt/suspect/${mountPoint}${2}/.mozilla/firefox" ]]; then
     echo "    Found where it is supposed to be." | ${teeArg}
```

2.1.3 Obtaining Microsoft Windows User Information

For Microsoft Windows users, obtaining their information involved a Python script which parses their NTUSER.DAT and SAM registry file. This file could potentially be in a non-standard place dependant on user settings. Therefore a simple BASH script is used to find these files.

```
               samRegKeyLoc=$(find ${driveRootPath} -type f -iname 'sam' | grep -i 'system32' | tail -n1)
               python /scripts/sysInfo/winUserInfo.py "${samRegKeyLoc}" > "/evidence/parsedData/${caseID}/
${dirExaminerName}/${dirCaseDate}/${mountPoint}Data/users.info"
               winUserFolders=$(find ${driveRootPath}/Documents\ and\ Settings/*/ -maxdepth 1 -iname
ntuser.dat 2>/dev/null | grep -v '/Default/' | grep -v '/Default User/' | grep -v '/Public/' | grep -v '/
All Users/')
               if [[ "${winUserFolders}" == "" ]]; then
                    echo " WARNING: No User Registry Keys Found on ${mountPoint}! No user specific
information will be collected." | ${teeArg}
```

The winUserInfo.py Python script called above grabs the Microsoft Windows user information using the SAM registry file. The NTUSER.DAT file is used to get the users local and roaming directories. This is where the users Firefox profiles will be stored and are parsed alongside the GNU/Linux partition profiles.

2.2 Acquisition Python Code

The Python scripting language is used to parse most of the important forensic information for the Web History Extraction Artefact Tool, all of these scripts are called by BASH scripts as required.

2.2.1 Microsoft Windows Operating System Information

Microsoft Windows operating system information is stored in the SOFTWARE registry file. This file contains a range of information valuable to an investigation such as the OS version, edition, registered owner and installation date.

```
# Open Registry Key
reg = Registry.Registry(sys.argv[1])

# Find Key Containing Windows Version
key = reg.open("Microsoft\Windows NT\CurrentVersion")

# Get Value of Keys
valPN = key.value("ProductName")
valEI = key.value("EditionId")
valCB = key.value("CurrentBuildNumber")
valID = key.value("InstallDate")
valRO = key.value("RegisteredOwner")
valRG = key.value("RegisteredOrganization")

# Convert UNIX EPOCH to HH:MM:SS DD/MM/YYYY UTC
installDate = datetime.datetime.utcfromtimestamp(valID.value()).strftime('%H:%M:%S, %d/%m/%Y')
```

2.2.2 Microsoft Windows User Roaming and Local Locations

As described above, the Microsoft Windows users store their Firefox profiles in the roaming and local locations. These locations can be changed from their default. To retrieve their locations from the registry, the following script is used.

```
# Open Registry Key
reg = Registry.Registry(sys.argv[1])

# Find Key Containing Windows Version
key = reg.open("Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders")

# Get Value of Keys
valAD = key.value(sys.argv[2])

# Return Values"
print "%s" % (valAD.value())
```

2.2.3 Microsoft Windows User Information

To obtain the information of Microsoft Windows users, the SAM file is parsed. This file is cleverly designed to store information in as little space as possible; therefore it is quite difficult to parse. To read the registry file itself, the Python module python-registry by William Ballenthin is used to obtain the hive information.

The data in the SAM registry file needs to be swapped from big endian to little endian; this is done with a simple function which gets every two HEX bytes of a string and adds them to the front of the new HEX string; this would obtain '`123456`' and return '`563412`'

```
def swapEnd(endToSwap):
    # Swap endianness of the hex (1234 becomes 3412)
    swappedEnd = ''
    for i in range(0, len(endToSwap), 2):
        swappedEnd = "%s%s" % (endToSwap[i:i+2], swappedEnd)
    return swappedEnd
```

The obtained strings from the registry will be seen as two bytes per single actual byte; i.e. the HEX 01 which is one byte, is seen within Python as '0' and '1' which is two bytes. To easily parse information a function is used to grab the correct bytes from the HEX code. This is done by

multiplying the initial byte number by two and then multiplying by two and adding a further two to the end number.

```
def selHex(string, start, end):
     # Returns selected hex
     hexStart = start*2
     hexEnd = end*2+2
     return string[hexStart:hexEnd]
```

Certain values within the SAM file can be converted straight from HEX into a value, such as the password reset time.

```
# Password reset time
rawPR = int(swapEnd(selHex(userF, 24, 31)), 16)
if rawPR != 0:
     datePR = datetime(1601,1,1) + timedelta(microseconds=rawPR/10)
else:
     datePR = 'Never'
```

However other values, such as the flag to indicate password requirement, need to be checked against pre-assigned values.

```
# Password exists
if int(selHex(userV, 172, 172)) == 14:
     passEx = 'Yes'
else:
     passEx = 'No'

# Password required to login
if int(userF [56*2]) == 1:
     passReq = 'Required'
else:
     passReq = 'Not Required'
```

2.2.4 Cache Rebuilding

Both the _CACHE_MAP_ and cache2 index files for Firefox contain a “dirty” flag which is set when the browser was shutdown in an unsafe or unclean manner, generally resulting in corrupt or unreadable cache data. This flag can be checked, on both files, with the following Python snippet.

```
    # Check for potential corruption (0 means clean shutdown, 1 means corruption or unclean
shutdown)
    dirtyFlag = struct.unpack('>L', header[8:12])[0]
    dirtyMap = ["The dirty flag was set. Firefox was not shut down cleanly and the collected data
may be unreliable or non-existent.", "The dirty flag was not set. The collected data is reliable."]
[dirtyFlag == 0]
```

The legacy Firefox Cache uses a _CACHE_MAP_ file along with _CACHE_00[123]_ files to store both data and metadata. The _CACHE_MAP file can be parsed to find the Firefox version. This map file also contains data which can be ran against multiple offsets and masks to obtain locations and file sizes. This information has been obtained directly from the Firefox source code and runs in the same fashion.

```
# Firefox Pre-Defined Masks/Offsets
resMask = 0x4C000000
locSelMask = 0x30000000
locSelOff = 28
extraBlockMask = 0x03000000
extraBlockOff = 24
blockNumMask = 0x00FFFFFF
fileGenMask = 0x000000FF
fileSizeMask = 0x00FFFF00
fileSizeOff = 8
fileResMask = 0x4F000000
```

These variables can be used to check that the current bucket contains a valid entry.

```
# Ensure record is valid (has Hash, metadata location and data location)
if int(rHash, 16) != 0 and not (rData & resMask) and not (rMeta & resMask):
     # Determine metadata location
     mLoc = (rMeta & locSelMask) >> locSelOff
     # External metadata file
```

The '`mLoc`' variable above will contain a value between 0 and 3, where 0 is an external file and 1-3 refer to _CACHE_00[123]_ respectively.

The other variables are used to determine the exact offset of these files where the data or metadata is located. This data is extracted in a similar manner to the cache2 files as illustrated in section 2.2.5.

The external cache data, depending on the Mozilla Firefox version, can be of two different directory structures. The tool determines where the external file lies with the following code.

```
# Firefox 2.0+
if mozVer == 'Mozilla 2.0 Firefox 4.0 (or above)':
     dExFile = "%s/Cache/%s/%s/%sd%02d" % (sys.argv[1], rHash[0], rHash[1:3], rHash[3:8], dGen)
# Firefox <2.0
else:
     dExFile = "%s/Cache/%sd%02d" % (sys.argv[1], rHash, dGen)
```

2.2.5 Cache2 Rebuilding

If the cache2 index file is clean, it should contain the filenames of each cache entry (this filename is a hash of the URL). These filenames are obtained and then the file is searched for and, if found, read with the following code.

```
# Get entry hash
indexHash = binascii.hexlify(indexE[:20]).upper()
# Determine entry file
indexEntry = "%s/cache2/entries/%s" % (sys.argv[1], indexHash)
# Check entry file exists
if os.path.isfile(indexEntry):
```

When reading the cache2 file, the data is stored before the metadata. The starting location of the metadata is stored at the very last 4 bytes of the file. This number, plus four bytes and the number of chunks doubled, is the starting location of the metadata for each file.

```
# Determine location of metadata (last 4 bytes contains this)
mLoc = struct.unpack('>L', entry[-4:])[0]

# Calclate number of chunks
mChunks = int(ceil(float(mLoc)/float(chunkSize)))
# Retrieve metadata
mData = entry[(mLoc+(mChunks*2)+4):]
```

This metadata contains information such as the fetch count, last modified time and last fetched time. The following regular expression obtains the URL (request string) and HTTP response header locations and then obtains the file names, removing any GET requests as they are not part of the file name. Null bytes in the response headers are delimiters, to make these PHP friendly they are replaced with a bespoke delimiter.

```
# URL
urlS = struct.unpack('>L', mData[24:28])[0]
# Retrieve Request and Response Strings
if '?' in  mData[28:(28+urlS)]:
     urlD = re.sub('.*(http[s]?://.*)\?.*', r'\1', mData[28:(28+urlS)])
     get = re.sub('.*http[s]?://.*(\?.*)', r'\1', mData[28:(28+urlS)])
else:
     urlD = re.sub('.*(http[s]?://.*)', r'\1', mData[28:(28+urlS)])
     get = 'No get data sent with the URL.'

# HTTP Response Headers
res = mData[(28+urlS)+1:].replace('\00', '#::#')
```

Sometimes data is sent across compressed within a GZIP container; the python-magic module is used to determine if the current data has a GZIP file header and attempts to unzip the data if this is true.

```
# Check if GZIP
gzipCheck = magic.from_buffer(data)
if 'gzip compressed data' in gzipCheck:
     # Make buffer data act like a file
     gzipIoData = StringIO(data)
     # Unzip the "file" (data)
     gzipData = gzip.GzipFile(fileobj=gzipIoData)
```

Sometimes the data cannot be unzipped, this appears to be because of malformed data as GZIP returns a CRC check failure and ZLIB returns a corrupt data error.

```
                              try:
                                   data = gzipData.read()
                              except:
                                   data = 'Unable to decompress this file. The CRC check probably
failed meaning this compressed data was corrupt or unreadable.'
```

2.2.6 Obtaining Downloaded Files

As downloaded files may be of interest to the examiner, the Python script which parses the places.sqlite file searches for the local file; if it is found then this file is copied to the evidence drive.

```
# Try to get local file if it still exists
downCopyFile = urllib.unquote(downFurList[1].replace("file:///", "").replace("c:/", "").replace("C:/", ""))
if os.path.isfile("%s/%s" % (sys.argv[4], downCopyFile)):
     print "  Retrieved downloaded file '%s'" % downCopyFile
     copyfile("%s/%s" % (sys.argv[4], downCopyFile), "%s/%s.%s" % (sys.argv[5], downList[0], downList[3]))
     downFile="%s#:#File Copied:#:#Yes (%s/%s.%s)" % (downFile, sys.argv[5], downList[0], downList[3])
else:
     downFile="%s#:#File Copied:#:#No" % downFile
```

2.2.7 Obtaining Web History

The sqlite database places.sqlite contains much of the information required for this tool. Both moz_places and moz_historyvisits can be used in conjunction with each other to obtain a list of the times and dates a single page was visited. The SQL in Python can be given a '?' to allow the code to later insert a value. This is used to allow the moz_historyvisits SQL again without the need for any change in the SQL.

```
# moz_places SQL
placeDate = "datetime(last_visit_date/1000000,'unixepoch','localtime')"
placeSelect = "id, title, url, visit_count, %s" % (placeDate)
placeSql = "SELECT %s FROM moz_places WHERE hidden = 0 AND visit_count > 0 ORDER BY last_visit_date DESC" % (placeSelect)
# moz_historyvisits SQL
histDate = "datetime(visit_date/1000000,'unixepoch','localtime')"
histSelect = "id, %s, visit_type, from_visit" % (histDate)
histSql = "SELECT %s FROM moz_historyvisits WHERE place_id = ? ORDER BY visit_date DESC" % (histSelect)
```

2.3 Viewing PHP Code

The resultant data is presented to the examiner in the form of a PHP file.

2.3.1 Parsing Output Files

Due to some URLs and/or names containing a '' ', the output files are stored with a delimiter of #:# or #'# to ensure that the data is correctly displayed. This has been chosen as it is very unlikely that a URL or name will contain this string.

```
                              $cookieDomainName=file_get_contents("/evidence/parsedData/
$openCaseName/$examPath/$datePath/$sdxData/$currUser/$allUserPathDir/$allMozProf/googleCookies/
$allCookiesDataDir");
                              $cookieDomainName=explode("#'#", $cookieDomainName)[1];
```

2.3.2 While Loops

To parse through all cases, drives, users, browsers and profiles the PHP system checks that directories exist and loop through each directory recursively to obtain all relevant data.

```
if ($allUserPathDir == "firefox") {
     if ($mozProfDir = opendir("/evidence/parsedData/$openCaseName/$examPath/$datePath/$sdxData/
$currUser/$allUserPathDir")) {
     while (false !== ($allMozProf = readdir($mozProfDir))) {
```

2.3.3 Correctly Displaying Images

Depending on the current item, a different image is displayed such as a Firefox emblem or a Tux penguin. This image is chosen by opening relevant files and checking for certain strings.

```
$currPartSysRaw=file_get_contents("/evidence/parsedData/$openCaseName/$examPath/$datePath/${partPath
[2]}.os");
$currOsCheck=trim("$currPartSysRaw");
if (stripos($currOsCheck, "windows")) {
     $osImg='ms';
} else if (stripos($currOsCheck, "linux")) {
     $osImg='linux';
} else {
     $osImg='unknown';
}
echo "<div class=\"sinPartSys\">
     <table>
     <tr>
          <th colspan='2'><img src='/images/$osImg-large.png'/></th></tr><tr>
```

2.3.4 Users Contain Data

Some users may not have any web browsing data associated with their account. For these users there is little point presenting the examiner with a link to view data as it will be blank. A similar method to 2.3.3 is used to only display links for users which contain valid information. Here, the link will only be displayed if the user has a directory on the evidence drive that contains data.

```
     if ($findNameDir = opendir("/evidence/parsedData/$openCaseName/$examPath/$datePath/${partPath
[2]}Data")) {
     while (false !== ($findName = readdir($findNameDir))) {
          if ($uName[1] == $findName) {
               echo "<tr><th colspan='2' class='histUser'><form action='http://wheat'
method='post'><input type='hidden' name='mainSection' value='history'><input type='hidden'
name='caseID' value='$openCaseName'><input type='hidden' name='dateID' value='$datePath'><input
type='hidden' name='examID' value='$examPath'><input type='hidden' name='driveID'
value='$drivePath'><input type='hidden' name='partID' value='$currPartNum'><input type='hidden'
name='username' value='$findName'><input type='hidden' name='histInfoType' value='sys'><input
type='submit' value='View History'></form></th></tr></table></div>";
          }
     }
     }
```

2.3.5 Google Analytics Cookies

Google Analytics cookies contain a wide range of information which is helpful to an examiner such as how many pages were viewed on a domain, or how the user found a certain website. As this is a service provided by Google to website owners, only the information owners are requesting are kept in these cookies. The Google Analytics page parses through these retrieved cookies and displays them in a human readable format.

```
               if ($currUTMZdata[0] == 'utmcsr') {
                    $utmzData=str_replace(array('(', ')'), '', $currUTMZdata[1]);
                    $zRow .= "<tr><th class='dash'>Last site used to access target site</th><td
class='dash'>$utmzData</td></tr>";
               } else if ($currUTMZdata[0] == 'utmccn') {
                    $utmzData=str_replace(array('(', ')'), '', $currUTMZdata[1]);
                    if ($utmzData == 'organic') {
                         $utmzData='Referred to site by search engine (e.g. Google or Bing)';
                    } else if ($utmzData == 'referral') {
                         $utmzData='Referred to site by a link on another site';
                    } else if ($utmzData == 'direct') {
                         $utmzData='URL was typed directly into the browser';
                    }
                    $zRow .= "<tr><th class='dash'>Ad campaign information</th><td class='dash'>
$utmzData</td></tr>";
```

2.3.6 Textual Searching

When a multitude of data is presented to an examiner, it is important that a search feature is implemented to allow efficient human interaction. A searchbar is atop all relevant web history modules.

```
if (! isset($_POST['histSearchTerm']) || stripos($searchLine, $_POST['histSearchTerm']) > 0 || trim($_POST['histSearchTerm']) == '') {
```

2.3.7 Date Searching

Date searches are performed in a similar way. The date is converted in to a number with the format 'YYYYMMDD' and data with a number greater than the minimum value and less than the maximum value is displayed.

```
if (! isset($_POST['disDate']) || $_POST['disDate'] == 'yes' || ($histDate >= $searchBegDate && $histDate <= $searchEndDate)) {
```

As the examiner may like to search without the date, another form has been added to allow the date function to be disabled and enabled. PHP does not allow nested forms, so an external form was added and the enable/disable button was told to use this form instead.

```
                $disButt="<input type='hidden' name='disDate' form='disForm' value='yes'><input
type='submit' form='disForm' value='Disable Date' class='searchButton' >";
```

2.3.8 Profile Searching

When there are multiple profiles associated with a user, the examiner may only wish to view a certain one.

```
    if (! isset ($_POST['profileSearch']) || $_POST['profileSearch'] == 'All' || "history.".$_POST
['profileSearch'] == $allMozProf) {
```

2.3.9 Protocol Searching

While searching for website, the examiner may find it useful to know which websites visited were viewed with or without SSL encryption. These protocols can be filtered using the search function.

```
if (! isset($_POST['searchHTTPS']) || $currHTTP != 'https:') {
if (! isset($_POST['searchHTTP']) || $currHTTP != 'http:') {
```

3.0 Conclusions and Recommendations

3.1 Completed Modules

While testing and viewing the results of the Web History Extraction Artefact Tool the cache2 data was rebuilt correctly and has been displayed in a manner advantageously to the examiner. The history, downloads and cookies are currently parsed as much as automation can be done. The examiner is given the full information and these modules are classed as fully completed as far as the tool is concerned.

3.2 Incomplete / Error Prone Modules

During the unit testing of the Mozilla Firefox cache, unexpected errors were encountered which resulted in much larger than expected volumes of data being written to the disk. This appears to be due to the HTTP headers of data and metadata overtaking valid references of data that should have been written. Further information regarding this software bug can be found alongside the Unit Testing report in section A020.

3.3 Potential Future Modules

The cache and cache2 modules of the tool can be further extended to use SED commands which can replace current paths within HTML documents with the local paths recovered from the cache. This would result in the HTML being displayed with the correct images and CSS that it was originally viewed with. This potential upgrade could be implemented in a similar manner that the current pages are currently implemented as.

As a further potential module, data recovered such as the retrieved download files could be viewed using the python-magic module and then displayed directly to the user if possible, for example if the resultant file is a graphical image.