Mario Barrenechea
CMPSCI365: Digital Forensics
2/8/10
Homework 1

On the Programming Assignment:

Run these commands:

```
$javac *.java
$java FFCacheParser.java <Absolute Path to Cache Directory>
```

---

1) **According to the Tucker decision, what are the definitions of "reasonable suspicion" and "probable cause"?**

   The definition of "reasonable suspicion" is: gauging the level of some activity occurring or heard about by others. Reasonable suspicion is the motive for believing that something is happening because of leading information that supports that motive.

   The definition of "probable cause" is similar to that of reasonable suspicion. Probable cause is having the intuition to identify that a crime has or had occurred in sight. If the law enforcement officer identifies that a crime has been committed, she can use probable cause to search without a warrant or seize evidence at the scene. It is a subjective term because the officer has to use her experience and intuition to detect that a crime has been committed, and it is often hard to justify probable cause when a non-obvious crime occurs.

2) **Tucker argued that even though the search of his home was permissible, the search of his computer was not according to the decision in US vs. Carey. What was the specific situation in Carey where a search was said to not be permissible? What was the court's decision regarding Tucker's claim and why it was the same or different from Carey?**

   The specific situation in US vs. Carey (*District of Kansas)* describes the case in where law enforcement officers suspected Mr. Carey of engaging in drug trafficking and possession of cocaine, marijuana, and hallucinogenic mushrooms. A search warrant was obtained from Carey, to which he readily consented to, and a seizure of these drugs including his two personal computers were taken as evidence from his apartment.

   Detective Lewis led the investigation of the seized evidence and took to identifying leading information about Mr. Carey's drugs on both of his computers. However, upon viewing certain image folders from them, he found a handful of images of child pornography, contraband indicating that Mr. Carey was in knowing possession of these images. In addition to the drug counts against Carey, several counts of possession of contraband were also counted against him, but Carey later testified that he believes that under the Fourth Amendment, Detective Lewis had no right of searching and seizing these images of contraband without probable cause to do so. [1]

The court's decision regarding the defendant Tucker and the prosecution rested on his knowledge of viewing child pornography on his personal computer [2]. Tucker admitted that he subscribed to such pornographic channels in which he was offered access to hundreds of images, which, Tucker claimed he never downloaded to his hard drive. In all of his browsing sessions, he testified that he always deleted them from his cache. The court's decision found him guilty of knowing possession of contraband, which is in contrast to US vs. Carey, where prosecution suppressed the charges of contraband counts against Carey because of unconstitutional search and seizure.

3) According to the Tucker decision, what is the "plain-view doctrine" for exceptions to search warrants? What specific conditions must be met?

The plain-view doctrine for exceptions to search warrants describes the circumstance that presents itself when suspicious activity or a crime is occurring within plain sight. Law enforcement officers are able to commit search and seizure when such activity is taking place within reasonable suspicion and above the 4th Amendment [3]. However, in order for an officer to identify items as evidence at a scene, he or she must meet conditions. First, the officer must already be a lawful presence in an area protected by the 4th Amendment, and second, the officer must identify the item in plain view and recognize it as contraband or evidence linking to a crime.

4) Ty Howard's article categorized six indications of contraband possession used by courts when deciding cases. What are they?

The six indications of contraband possession used by courts to decide cases are as follows:

a) The defendant's knowledge of the contraband
b) The defendant's destruction of the contraband
c) The defendant's manipulation of the contraband
d) The defendant's intention to seek out the contraband
e) How much contraband is found
f) Other relevant evidence (other)

References:

[1]: "98-3077 -- U.S. v. Carey -- 04/14/1999." 2/10/2010 <http://ca10.washburnlaw.edu/cases/1999/04/98-3077.htm>.

[2]: Howard, Ty E. "DON''T CACHE OUT YOUR CASE: PROSECUTING CHILD PORNOGRAPHY POSSESSION LAWS BASED ON IMAGES LOCATED IN TEMPORARY INTERNET FILES." .

SVN COMMANDS:

SVN Commands:

svn info:

```
Path: .
URL:
svn+ssh://mbarrene@elnux7.cs.umass.edu/courses/cs300/cs365/mbarrene/work/homeworkSVN/trunk/hws/src/hw
1
Repository Root: svn+ssh://mbarrene@elnux7.cs.umass.edu/courses/cs300/cs365/mbarrene/work/homeworkSVN
Repository UUID: 0d641f17-83e7-4f37-b317-d3728eef71d6
Revision: 25
Node Kind: directory
Schedule: normal
Last Changed Author: mbarrene
Last Changed Rev: 25
Last Changed Date: 2010-02-16 22:20:56 -0500 (Tue, 16 Feb 2010)
```

svn log:

```
r25 | mbarrene | 2010-02-16 22:20:56 EST

(CDB): Code Complete. Still questionable (buggy), but it is the final version.
------------------------------------------------------------------------
r24 | mbarrene | 2010-02-16 18:39:01 EST

(CDB): Added boolean flags to CacheRecord to determine whether data/metadata entries were parsed correctly. The last big step is
to refactor my parsing to include String formatting, and then testing the program for scalability!
------------------------------------------------------------------------
r23 | mbarrene | 2010-02-16 18:25:39 EST
------------------------------------------------------------------------
r22 | mbarrene | 2010-02-16 00:26:01 EST

(CDP): Working along, now onto Metadata parsing!
------------------------------------------------------------------------
r21 | mbarrene | 2010-02-15 18:27:36 EST

(CDP): More changes and refactoring to the code... Next steps should be relatively simple.
------------------------------------------------------------------------
r20 | mbarrene | 2010-02-15 15:51:07 EST

(CDP): Introducing CacheFile (Java Enum) and CacheRecord (Java class) to help store and organize important info for each record
read in..
------------------------------------------------------------------------
r19 | mbarrene | 2010-02-14 22:02:30 EST

(CDP): Introduced doParse(), which is the high level template for parsing the Cache Map and obtaining important and leading data.
------------------------------------------------------------------------
r18 | mbarrene | 2010-02-14 01:48:52 EST

(CDP): Argument-checking module is working properly...Now onto bigger business.
------------------------------------------------------------------------
r17 | mbarrene | 2010-02-13 17:25:22 EST
------------------------------------------------------------------------
```

```
r16 | mbarrene | 2010-02-13 17:23:56 EST

Code Development (CDP): Working on argument-checking module for FFCacheParser
------------------------------------------------------------------------
r15 | mbarrene | 2010-02-13 15:48:25 EST
------------------------------------------------------------------------
r14 | mbarrene | 2010-02-13 15:46:07 EST
------------------------------------------------------------------------
r13 | mbarrene | 2010-02-13 15:44:40 EST
------------------------------------------------------------------------
r12 | mbarrene | 2010-02-12 17:44:45 EST

Code setup (CS): Added hw1 package to forensicsHW source folder.
------------------------------------------------------------------------
r11 | mbarrene | 2010-01-26 09:07:34 EST

(CD): HW0 Done.
------------------------------------------------------------------------
r10 | mbarrene | 2010-01-26 08:00:37 EST

(CDB): Error in HW Assignment. Do not print char 127.
------------------------------------------------------------------------
r9 | mbarrene | 2010-01-24 23:56:03 EST

(DONE): hw0

------------------------------------------------------------------------
r8 | mbarrene | 2010-01-24 23:48:42 EST
------------------------------------------------------------------------
r7 | mbarrene | 2010-01-24 23:34:42 EST

(CDB): Finished with code, now onto Debugging...
------------------------------------------------------------------------
r6 | mbarrene | 2010-01-24 16:59:04 EST

(CDP): Finished dumpHex(), now onto the hard part: dumpStrings()....
------------------------------------------------------------------------
r5 | mbarrene | 2010-01-24 11:55:58 EST

refactored...
------------------------------------------------------------------------
r4 | mbarrene | 2010-01-24 11:51:28 EST
------------------------------------------------------------------------
r3 | mbarrene | 2010-01-24 11:50:53 EST

Initial import.
```

svn blame:

```
umass-952-123:hw1 mbarrenecheajr$ svn blame *.java
mbarrene@elnux7.cs.umass.edu's password:
    20   mbarrene package hw1;
    20   mbarrene
    20   mbarrene import java.io.File;
    20   mbarrene
    20   mbarrene /**
    20   mbarrene  * CacheFile Enum improves clarity and efficiency when determining cache file
    20   mbarrene  * sizes and filenames.
    20   mbarrene  *
    20   mbarrene  * @author mbarrenecheajr
    20   mbarrene  */
    20   mbarrene public enum CacheFile {
    20   mbarrene
    20   mbarrene         CACHE_MAP(128, "_CACHE_MAP_"),
    20   mbarrene         CACHE_001(256, "_CACHE_001_"),
    20   mbarrene         CACHE_002(1024, "_CACHE_002_"),
    20   mbarrene         CACHE_003(4096, "_CACHE_003_"),
    20   mbarrene         EXTERNAL(-1, "EXTERNAL"); //Entirely Mutable
    20   mbarrene
    20   mbarrene         //Members
    20   mbarrene
    21   mbarrene         /** Member for holding cache block size. **/
    21   mbarrene         private int cacheBlockSize_;
    21   mbarrene         /** Member for holding cache file size (as specified by the File object) **/
    21   mbarrene         private long cacheFileSize_;
    20   mbarrene         /** Member for holding the cache file name. **/
    20   mbarrene         private String cacheFileName_;
    20   mbarrene         /** Member for holding the cache File Java object. **/
    20   mbarrene         private File cacheFileObject_;
    21   mbarrene
    20   mbarrene
    20   mbarrene         // Getters
    21   mbarrene         protected int getCacheBlockSize() {return this.cacheBlockSize_;}
    21   mbarrene         protected long getCacheFileSize() {return this.cacheFileSize_;}
    20   mbarrene         protected String getCacheFileName() {return this.cacheFileName_;}
    20   mbarrene         protected File getCacheFileObject(){ return this.cacheFileObject_;}
    21   mbarrene
    20   mbarrene
    20   mbarrene         // Setters
    20   mbarrene         protected void setCacheFileObject(File f){this.cacheFileObject_ = f;}
    21   mbarrene         protected void setCacheFileSize(long l){this.cacheFileSize_ = l;}
    20   mbarrene         protected boolean setExternalFileSize(int size) {
    20   mbarrene                 if (this == CacheFile.EXTERNAL) {
    20   mbarrene                         this.cacheFileSize_ = size;
    20   mbarrene                         return true;
    20   mbarrene                 }
    20   mbarrene                 return false;
    20   mbarrene         }
    20   mbarrene         protected boolean setExternalFileName(String name){
    20   mbarrene                 if (this == CacheFile.EXTERNAL){
    20   mbarrene                         this.cacheFileName_ = name;
    20   mbarrene                         return true;
    20   mbarrene                 }
    20   mbarrene                 return false;
    20   mbarrene         }
    25   mbarrene
    25   mbarrene         /**
    25   mbarrene          * This function determines if the underlying
    25   mbarrene          * CacheFile is CACHE 001, 002, or 003.
    25   mbarrene          * @return
    25   mbarrene          */
    25   mbarrene         protected boolean isADefinedCacheFile(){
    25   mbarrene                 return (this == CACHE_001 || this == CACHE_002 || this == CACHE_003);}
    20   mbarrene
    20   mbarrene         // Cannot be instantiated!
    21   mbarrene         private CacheFile(int blocksize, String filename) {
    21   mbarrene                 this.cacheBlockSize_ = blocksize;
    20   mbarrene                 this.cacheFileName_ = filename;
    20   mbarrene         }
    20   mbarrene }
mbarrene@elnux7.cs.umass.edu's password:
    20   mbarrene package hw1;
    20   mbarrene
    23   mbarrene import java.util.Date;
    23   mbarrene
    20   mbarrene /**
    20   mbarrene  * Object Placeholder for Important Cache Record information including (but not limited to):
    20   mbarrene  *
    20   mbarrene  * 1) Data Location, Size, and File
    20   mbarrene  * 2) Metadata Location, Size, and File
    20   mbarrene  * 3) Metadata slack and hex dump
    20   mbarrene  *
```

```
20   mbarrene  * @author mbarrenecheajr
20   mbarrene  *
20   mbarrene  */
20   mbarrene public class CacheRecord {
20   mbarrene
20   mbarrene
20   mbarrene         /*** Members ***/
20   mbarrene         private int cacheRecordId_ = 0;
24   mbarrene         private boolean dataEntryParsed_ = false;
24   mbarrene         private boolean metadataEntryParsed_ = false;
20   mbarrene
20   mbarrene         //Data
20   mbarrene         private CacheFile dataCacheFile_ = null;
20   mbarrene         private String dataHexString_ = null;
20   mbarrene         private String dataBinaryString_ = null;
23   mbarrene         private Integer dataByteInteger_ = null;
20   mbarrene         private int dataBlockNumber_ = 0;
20   mbarrene         private int dataBlockCount_ = 0;
20   mbarrene         private int metadataBlockCount_ = 0;
25   mbarrene         private long dataFileOffset_ = 0;
23   mbarrene         private int dataObjectSize_ = 0;
25   mbarrene         private String data_ = null;
20   mbarrene
20   mbarrene         //Metadata
20   mbarrene         private CacheFile metadataCacheFile_ = null;
20   mbarrene         private String metadataHexString_ = null;
20   mbarrene         private String metadataBinaryString_ = null;
23   mbarrene         private Integer metadataByteInteger_ = null;
20   mbarrene         private int metadataBlockNumber_ = 0;
20   mbarrene         private int metadataFileOffset_ = 0;
23   mbarrene         private int metadataFetchCount_ = 0;
23   mbarrene         private Date metadataLastFetched_ = null;
23   mbarrene         private Date metadataLastModified_ = null;
23   mbarrene         private Date metadataExpirationTime_ = null;
23   mbarrene         private int metadataSize_ = 0;
23   mbarrene         private int metadataUrlSize_ = 0;
23   mbarrene         private String metadataUrl_ = null;
23   mbarrene         private String metadata_ = null;
23   mbarrene         private String metadataSlack_ = null;
25   mbarrene         private boolean metadataParsed_ = false;
20   mbarrene
20   mbarrene         /*** Getters ***/
20   mbarrene         protected int getCacheRecordId(){ return this.cacheRecordId_;}
24   mbarrene         protected boolean getDataEntryParsed(){ return this.dataEntryParsed_;}
24   mbarrene         protected boolean getMetadataEntryParsed(){ return this.metadataEntryParsed_;}
25   mbarrene         protected boolean getMetadataParsed(){ return this.metadataParsed_;}
20   mbarrene
20   mbarrene         //Data
20   mbarrene         protected CacheFile getDataCacheFile(){ return this.dataCacheFile_;}
20   mbarrene         protected String getDataHexString() {return dataHexString_;}
20   mbarrene         protected String getDataBinaryString() {return dataBinaryString_;}
20   mbarrene         protected int getDataBlockNumber() {return dataBlockNumber_;}
20   mbarrene         protected int getDataBlockCount() {return dataBlockCount_;}
25   mbarrene         protected long getDataFileOffset() {return dataFileOffset_;}
23   mbarrene         protected Integer getDataByteInteger(){ return dataByteInteger_;}
23   mbarrene         protected int getDataObjectSize(){ return this.dataObjectSize_;}
25   mbarrene         protected String getData(){ return this.data_;}
20   mbarrene
20   mbarrene         //Metadata
20   mbarrene         protected CacheFile getMetadataCacheFile(){ return this.metadataCacheFile_;}
20   mbarrene         protected String getMetadataHexString() {return metadataHexString_;}
20   mbarrene         protected String getMetadataBinaryString() {return metadataBinaryString_;}
20   mbarrene         protected int getMetadataBlockNumber() {return metadataBlockNumber_;}
20   mbarrene         protected int getMetadataBlockCount() {return metadataBlockCount_;}
20   mbarrene         protected int getMetadataFileOffset() {return metadataFileOffset_;}
23   mbarrene         protected Integer getMetadataByteInteger(){ return metadataByteInteger_;}
```

```
    23   mbarrene        protected int getMetadataFetchCount(){ return this.metadataFetchCount_;}
    23   mbarrene        protected Date getMetadataLastFetched(){ return this.metadataLastFetched_;}
    23   mbarrene        protected Date getMetadataLastModified(){ return this.metadataLastModified_;}
    23   mbarrene        protected Date getMetadataExpirationTime(){ return this.metadataExpirationTime_;}
    23   mbarrene        protected int getMetadataSize(){ return this.metadataSize_; }
    23   mbarrene        protected int getMetadataUrlSize(){ return this.metadataUrlSize_;}
    23   mbarrene        protected String getMetadataUrl(){ return this.metadataUrl_;}
    23   mbarrene        protected String getMetadata(){ return this.metadata_;}
    23   mbarrene        protected String getMetadataSlack(){ return this.metadataSlack_;}
    20   mbarrene
    20   mbarrene        /*** Setters ****/
    24   mbarrene        protected void setDataEntryParsed(boolean b){ this.dataEntryParsed_ = b;}
    24   mbarrene        protected void setMetadataEntryParsed(boolean b){ this.metadataEntryParsed_ = b;}
    25   mbarrene        protected void setMetadataParsed(boolean b){ this.metadataParsed_ = b;}
    20   mbarrene
    20   mbarrene        //Data
    20   mbarrene        protected void setDataCacheFile(CacheFile f){ this.dataCacheFile_ = f;}
    20   mbarrene        protected void setDataHexString(String dataHexString) {this.dataHexString_ = dataHexString;}
    20   mbarrene        protected void setDataBinaryString(String dataBinaryString) {this.dataBinaryString_ = dataBinaryString;}
    20   mbarrene        protected void setDataBlockNumber(int dataBlockNumber_) {this.dataBlockNumber_ = dataBlockNumber_;}
    20   mbarrene        protected void setDataBlockCount(int dataBlockCount_) {this.dataBlockCount_ = dataBlockCount_;}
    25   mbarrene        protected void setDataFileOffset(long offset){ this.dataFileOffset_ = offset;}
    23   mbarrene        protected void setDataByteInteger(Integer byteint){ this.dataByteInteger_ = byteint;}
    23   mbarrene        protected void setDataObjectSize(int i){ this.dataObjectSize_ = i;}
    25   mbarrene        protected void setData(String s){ this.data_ = s;}
    20   mbarrene
    20   mbarrene        //Metadata
    20   mbarrene        protected void setMetadataCacheFile(CacheFile f){ this.metadataCacheFile_ = f;}
    20   mbarrene        protected void setMetadataBinaryString(String metadataBinaryString) {this.metadataBinaryString_ =
metadataBinaryString;}
    20   mbarrene        protected void setMetadataHexString(String metadataHexString) {this.metadataHexString_ =
metadataHexString;}
    20   mbarrene        protected void setMetadataBlockNumber(int metadataBlockNumber_) {this.metadataBlockNumber_ =
metadataBlockNumber_;}
    20   mbarrene        protected void setMetadataBlockCount(int metadataBlockCount_) {this.metadataBlockCount_ =
metadataBlockCount_;}
    20   mbarrene        protected void setMetadataFileOffset(int metadataOffset){ this.metadataFileOffset_ = metadataOffset;}
    23   mbarrene        protected void setMetadataByteInteger(Integer byteint){ this.metadataByteInteger_ = byteint;}
    23   mbarrene        protected void setMetadataFetchCount(int i){ this.metadataFetchCount_ = i;}
    23   mbarrene        protected void setMetadataLastFetched(Date d){ this.metadataLastFetched_ = d;}
    23   mbarrene        protected void setMetadataLastModified(Date d){ this.metadataLastModified_ = d;}
    23   mbarrene        protected void setMetadataExpirationTime(Date d){ this.metadataExpirationTime_ = d;}
    23   mbarrene        protected void setMetadataSize(int i){ this.metadataSize_ = i;}
    23   mbarrene        protected void setMetadataUrlSize(int i){ this.metadataUrlSize_ = i;}
    23   mbarrene        protected void setMetadataUrl(String s){ this.metadataUrl_ = s;}
    23   mbarrene        protected void setMetadata(String s){ this.metadata_ = s;}
    23   mbarrene        protected void setMetadataSlack (String s){ this.metadataSlack_ = s;}
    20   mbarrene
    23   mbarrene
    23   mbarrene
    23   mbarrene        //Constructor
    20   mbarrene        private CacheRecord(int id){
    20   mbarrene                this.cacheRecordId_ = id;
    20   mbarrene        }
    20   mbarrene
    20   mbarrene        /**
    20   mbarrene         * Factory Method for creating new CacheRecords.
    20   mbarrene         * @return
    20   mbarrene         */
    20   mbarrene        public static CacheRecord createCacheRecord(int id){return new CacheRecord(id);}
    20   mbarrene
    20   mbarrene }
mbarrene@elnux7.cs.umass.edu's password:
    12   mbarrene package hw1;
    19   mbarrene
    16   mbarrene import java.io.File;
```

```
16  mbarrene import java.io.FileInputStream;
19  mbarrene import java.io.FileNotFoundException;
23  mbarrene import java.io.FileWriter;
19  mbarrene import java.io.IOException;
19  mbarrene import java.nio.ByteBuffer;
23  mbarrene import java.util.Date;
12  mbarrene
19  mbarrene /**
19  mbarrene  *
19  mbarrene  * This class is responsible for parsing the Firefox cache by reading the cachemap, determining the size,
19  mbarrene  * location, and metadata for each record in the same directory, and finally printing out the results in
19  mbarrene  * a readable fashion.
19  mbarrene  *
19  mbarrene  * @author mbarrenecheajr
19  mbarrene  *
19  mbarrene  */
25  mbarrene
14  mbarrene public class FFCacheParser {
12  mbarrene
25  mbarrene         /******** FFCacheParser class definition *******/
25  mbarrene
25  mbarrene         private int recordCount_ = 0;
25  mbarrene         private File outDirectory_ = null;
25  mbarrene
25  mbarrene         /** Cache Files **/
25  mbarrene         private CacheFile cachemap_ = null;
25  mbarrene         private CacheFile cache001_ = null;
25  mbarrene         private CacheFile cache002_ = null;
25  mbarrene         private CacheFile cache003_ = null;
25  mbarrene
25  mbarrene         /** Cache File Setters **/
25  mbarrene         private void setCacheMap(CacheFile f){ this.cachemap_ = f;}
25  mbarrene         private void setCache001(CacheFile f){ this.cache001_ = f;}
25  mbarrene         private void setCache002(CacheFile f){ this.cache002_ = f;}
25  mbarrene         private void setCache003(CacheFile f){ this.cache003_ = f;}
25  mbarrene
25  mbarrene         /** Cache File Getters **/
25  mbarrene         private CacheFile getCacheMap(){ return this.cachemap_;}
25  mbarrene         private CacheFile getCache001(){ return this.cache001_;}
25  mbarrene         private CacheFile getCache002(){ return this.cache002_;}
25  mbarrene         private CacheFile getCache003(){ return this.cache003_;}
25  mbarrene
25  mbarrene         //Other
25  mbarrene         private File getOutDirectory(){ return this.outDirectory_;}
25  mbarrene
25  mbarrene         private FFCacheParser(){
25  mbarrene
25  mbarrene                 this.outDirectory_ = new File("out");
25  mbarrene                 this.outDirectory_.mkdir();
25  mbarrene         }
25  mbarrene
14  mbarrene         public static void main (String args []){
14  mbarrene
19  mbarrene                 FFCacheParser parser = new FFCacheParser();
19  mbarrene
19  mbarrene                 //First, check the Program Argument and setup the Parser
19  mbarrene                 if(parser.checkArgs(args) == false)
14  mbarrene                         System.exit(1);
19  mbarrene                 else
19  mbarrene                         parser.doParse();
14  mbarrene         }
14  mbarrene
25  mbarrene
23  mbarrene         /**doParse():
23  mbarrene          *
19  mbarrene          * This function is the high-level construct for doing several things in order as follows:
   19   mbarrene          * <p>
   22   mbarrene          * 1) Read in one record at a time, parsing both data and metadata entries and storing relevant information as
appropriate.
   19   mbarrene          * <p>
   22   mbarrene          * 2) Parse the Cache Files for both data and metadata from a record, the first 8 bytes from the data entry, and
the Metadata dump from the Metadata entry.
   19   mbarrene          * <p>
   23   mbarrene          * 3) Report and Write to files in a subdirectory called "out"
   19   mbarrene          */
   19   mbarrene         private void doParse() {
   19   mbarrene
   19   mbarrene                 try{
   19   mbarrene
   20   mbarrene                         FileInputStream fs = new FileInputStream(this.getCacheMap().getCacheFileObject());
   19   mbarrene
   19   mbarrene                         //Skip the first 276 bytes first
   22   mbarrene                         fs.skip(276);
   19   mbarrene
   19   mbarrene                         //While there are more bits and bytes to read in (CacheMap)
   19   mbarrene                         while(fs.available() > 0){
   19   mbarrene
   20   mbarrene                                 CacheRecord record = CacheRecord.createCacheRecord(this.recordCount_);
   19   mbarrene
   19   mbarrene                                 //Create a byte array of 128 bits so we can read in relevant information
   19   mbarrene                                 byte nextBytes [] = new byte [(128/Byte.SIZE)];
   19   mbarrene                                 fs.read(nextBytes);
   19   mbarrene
   24   mbarrene                                 /** Parse the Data and the Metadata Entries to obtain important information
   24   mbarrene                                 about the Web objects. Also, check if these entries were parsed correctly **/
   24   mbarrene                                 record.setDataEntryParsed(parseDataEntry(record, nextBytes, 8));
   24   mbarrene                                 record.setMetadataEntryParsed(parseMetadataEntry(record, nextBytes,
12));
   23   mbarrene
   24   mbarrene                                 if(record.getDataEntryParsed() == false && record.getMetadataEntryParsed()
== false){
   22   mbarrene
   22   mbarrene                                         this.recordCount_++;
   22   mbarrene                                         continue;
   21   mbarrene                                 }
   19   mbarrene
   22   mbarrene                                 //Parse the first 8 bytes out of the cache file
   25   mbarrene                                 if((record.getDataCacheFile() != null) && (record.getDataCacheFile() !=
CacheFile.EXTERNAL) && (record.getDataEntryParsed() == true))
   25   mbarrene                                         record.setData(this.parseDataFromCacheFile(record));
   22   mbarrene
   23   mbarrene                                 //Parse the Metadata Header
   25   mbarrene                                 if((record.getMetadataCacheFile() != null) &&
(record.getMetadataCacheFile() != CacheFile.EXTERNAL) && (record.getMetadataEntryParsed() == true))
   25   mbarrene
            record.setMetadataParsed(this.parseMetadataFromCacheFile(record));
   22   mbarrene
   25   mbarrene                                 //Finally, write everything to the record file under the subdirectory "out"
   25   mbarrene                                 this.writeDataToFile(record);
   23   mbarrene                                 this.recordCount_++;
   19   mbarrene                         }
   19   mbarrene
   19   mbarrene                         fs.close();
   20   mbarrene                 } catch(IOException io){io.printStackTrace();}
   19   mbarrene
   20   mbarrene         }//end doParse();
   19   mbarrene
   22   mbarrene         private boolean parseMetadataFromCacheFile(CacheRecord record) throws IOException{
   22   mbarrene
   23   mbarrene                 Hexdump.dumpHex(record.getMetadataCacheFile().getCacheFileObject().getAbsolutePath(), 8, 8,
(long)record.getMetadataFileOffset());
   23   mbarrene
```

```
22  mbarrene        FileInputStream fs = new FileInputStream(record.getMetadataCacheFile().getCacheFileObject());
22  mbarrene        fs.skip(record.getMetadataFileOffset());
22  mbarrene
23  mbarrene        //Calculate the Limit to what we are reading from the metadata entry in this cache file
22  mbarrene        int limit = (record.getMetadataBlockCount() * record.getMetadataCacheFile().getCacheBlockSize());
23  mbarrene
23  mbarrene        //Counts the Number of Bytes up to important points in parsing
22  mbarrene        int byteCount = 0;
23  mbarrene        byte nextBytes [] = new byte [4];
23  mbarrene
23  mbarrene        /**Reading in Constant (00 01 00 0C) **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene
23  mbarrene        //Variables
23  mbarrene        Integer byteInt = ByteBuffer.wrap(nextBytes).getInt();
23  mbarrene        String binaryString = Integer.toBinaryString(byteInt);
23  mbarrene
23  mbarrene        //Verifying Constant
23  mbarrene        if(byteInt != 0x0001000C){
25  mbarrene                //System.out.println("ERROR: Header Constant is inconsistent. Skipping...");
23  mbarrene                return false;
22  mbarrene        }
22  mbarrene
23  mbarrene        /** Reading in location **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene
23  mbarrene        /** Reading in Fetch Count **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene        byteInt = ByteBuffer.wrap(nextBytes).getInt();
23  mbarrene        binaryString = Integer.toBinaryString(byteInt);
23  mbarrene        record.setMetadataFetchCount(Integer.parseInt(binaryString, 2));
23  mbarrene
23  mbarrene        /** Reading in Last Fetched Date **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene        long byteLong = ByteBuffer.wrap(nextBytes).getInt();
23  mbarrene        Date lastFetched = new Date(byteLong);
23  mbarrene        record.setMetadataLastFetched(lastFetched);
23  mbarrene
23  mbarrene        /** Reading in Last Modified Date **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene        byteLong = ByteBuffer.wrap(nextBytes).getInt();
23  mbarrene        Date lastModified = new Date(byteLong);
23  mbarrene        record.setMetadataLastModified(lastModified);
23  mbarrene
23  mbarrene        /** Reading in Expiration Time **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene        byteLong = ByteBuffer.wrap(nextBytes).getInt();
23  mbarrene        Date expiration = new Date(byteLong);
23  mbarrene        record.setMetadataExpirationTime(expiration);
23  mbarrene
23  mbarrene        /** Reading in Data Object Size **/
23  mbarrene        fs.read(nextBytes);
23  mbarrene        byteCount += nextBytes.length;
23  mbarrene        byteInt = ByteBuffer.wrap(nextBytes).getInt();
23  mbarrene        binaryString = Integer.toBinaryString(byteInt);
23  mbarrene        int dataObjectSize = Integer.parseInt(binaryString, 2);
23  mbarrene        record.setDataObjectSize(dataObjectSize);
23  mbarrene
23  mbarrene        /** Reading in Metadata URL Size **/
23  mbarrene        fs.read(nextBytes);
```

```
  23   mbarrene                    byteCount += nextBytes.length;
  23   mbarrene                    byteInt = ByteBuffer.wrap(nextBytes).getInt();
  23   mbarrene                    binaryString = Integer.toBinaryString(byteInt);
  23   mbarrene                    int metadataUrlSize = Integer.parseInt(binaryString, 2);
  23   mbarrene                    record.setMetadataUrlSize(metadataUrlSize);
  23   mbarrene
  23   mbarrene                    /** Reading in Metadata Size **/
  23   mbarrene                    fs.read(nextBytes);
  23   mbarrene                    byteCount += nextBytes.length;
  23   mbarrene                    byteInt = ByteBuffer.wrap(nextBytes).getInt();
  23   mbarrene                    binaryString = Integer.toBinaryString(byteInt);
  23   mbarrene                    int metadataSize = Integer.parseInt(binaryString, 2);
  23   mbarrene                    record.setMetadataSize(metadataSize);
  23   mbarrene
  23   mbarrene                    /** Now, let's read in the Metadata URL **/
  23   mbarrene                    nextBytes = new byte [metadataUrlSize];
  23   mbarrene                    fs.read(nextBytes);
  23   mbarrene                    byteCount += nextBytes.length;
  23   mbarrene                    String metadataUrl = new String(nextBytes);
  23   mbarrene                    record.setMetadataUrl(metadataUrl);
  23   mbarrene
  23   mbarrene                    /** Grab the Metadata itself **/
  23   mbarrene                    String metadata =
Hexdump.dumpStrings(record.getMetadataCacheFile().getCacheFileObject().getAbsolutePath(),
  23   mbarrene                                    metadataSize, 64, (record.getMetadataFileOffset() + byteCount));
  23   mbarrene                    record.setMetadata(metadata);
  23   mbarrene                    byteCount += metadataSize;
  23   mbarrene
  23   mbarrene                    //This should be a reasonable number (i.e. non-negative)
  23   mbarrene                    int slackSize = limit - byteCount;
  23   mbarrene                    assert(slackSize > 0 == true);
  23   mbarrene
  23   mbarrene                    /** Finally, print out the slack data as a hex dump **/
  23   mbarrene                    String metadataSlack =
Hexdump.dumpHex(record.getMetadataCacheFile().getCacheFileObject().getAbsolutePath(),
  23   mbarrene                                    slackSize, 8, (record.getMetadataFileOffset() + byteCount));
  23   mbarrene
  23   mbarrene                    record.setMetadataSlack(metadataSlack);
  23   mbarrene
  23   mbarrene                    fs.close();
  22   mbarrene                    return true;
  22   mbarrene
  22   mbarrene            }
  22   mbarrene
  23   mbarrene            /** Get the First 8 Bytes from the Hex Dump of the Data Cache File **/
  23   mbarrene            private String parseDataFromCacheFile(CacheRecord record) throws IOException{
  22   mbarrene
  23   mbarrene                    return Hexdump.dumpHex(record.getDataCacheFile().getCacheFileObject().getAbsolutePath(), 8, 8,
(long)record.getDataFileOffset());
  22   mbarrene            }
  22   mbarrene
  20   mbarrene            /**
  20   mbarrene             * This function will modify the current CacheRecord for MetaData information.
  20   mbarrene             * @param record
  20   mbarrene             */
  23   mbarrene            private boolean parseMetadataEntry(CacheRecord record, byte bytes [], int index){
  20   mbarrene
  20   mbarrene                    //Grab the 32 bits that represents the metadata or data entry for this 128-bit record.
  20   mbarrene                    Integer byteInt = ByteBuffer.wrap(bytes, index, 4).getInt();
  20   mbarrene                    String hexString = Integer.toHexString(byteInt);
  20   mbarrene                    String binaryString = Integer.toBinaryString(byteInt);
  20   mbarrene
  23   mbarrene                    record.setMetadataByteInteger(byteInt);
  20   mbarrene                    record.setMetadataHexString(hexString);
  20   mbarrene                    record.setMetadataBinaryString(binaryString);
  20   mbarrene
```

```
   20   mbarrene                //Error Checking
   20   mbarrene                if(binaryString.length() != 32 || hexString.length() != 8){
   25   mbarrene                        //System.out.println("Record #" + this.recordCount_ + " is inconsistent with Hex and
Binary data (Metadata Entry). Skipping...");
   20   mbarrene                        return false;
   20   mbarrene                }
   20   mbarrene
   20   mbarrene                //Trim the Binary String and determine the cache file pertaining to this 128-bit record.
   20   mbarrene                String temp = binaryString.substring(2,4);
   20   mbarrene                int cacheFile = Integer.parseInt(temp, 2);
   20   mbarrene
   20   mbarrene                CacheFile cf = null;
   20   mbarrene                switch(cacheFile){
   20   mbarrene
   20   mbarrene                case 0: cf = CacheFile.EXTERNAL; break;
   20   mbarrene                case 1: cf = CacheFile.CACHE_001; break;
   20   mbarrene                case 2: cf = CacheFile.CACHE_002; break;
   20   mbarrene                case 3: cf = CacheFile.CACHE_003; break;
   21   mbarrene                default: System.out.println("ERROR: Unrecognized Cache File Number:
FFCacheParser.doParse().parseMetadataEntry()");
   20   mbarrene
   20   mbarrene                }
   20   mbarrene
   20   mbarrene                record.setMetadataCacheFile(cf);
   20   mbarrene
   20   mbarrene                //Then, trim the Binary String and determine the number of cache blocks that this 128-bit record
occupies.
   20   mbarrene                temp = binaryString.substring(6,8);
   20   mbarrene                int numCacheBlocks = 1 + Integer.parseInt(temp, 2);
   20   mbarrene                record.setMetadataBlockCount(numCacheBlocks);
   20   mbarrene
   20   mbarrene                //If not an external file, trim the Binary String and determine the cache block number of this 128-
bit record.
   20   mbarrene                int cacheBlockNum = 0;
   20   mbarrene                if(cf != CacheFile.EXTERNAL){
   20   mbarrene
   20   mbarrene                        temp = binaryString.substring(8, 32);
   20   mbarrene                        cacheBlockNum = Integer.parseInt(temp, 2);
   20   mbarrene                        record.setMetadataBlockNumber(cacheBlockNum);
   22   mbarrene
   25   mbarrene
   25   mbarrene
   22   mbarrene                        //Finally, determine the offset in the cache file wherein the data and metadata for this
128-bit record resides.
   22   mbarrene                        int offset = cf.getCacheBlockSize() * cacheBlockNum + 4096;
   22   mbarrene                        record.setMetadataFileOffset(offset);
   22   mbarrene
   25   mbarrene                        if(offset > 0 && offset < cf.getCacheFileSize())
   22   mbarrene                                return true;
   22   mbarrene                        else{
   25   mbarrene                                //System.out.println("ERROR: Offset is greater than the size of the Cache File
(Data Entry). Skipping...");
   22   mbarrene                                return false;
   22   mbarrene                        }
   22   mbarrene
   20   mbarrene                }
   20   mbarrene
   22   mbarrene                //External Files: No more work to be done; finish parsing.
   23   mbarrene                else
   21   mbarrene                        return true;
   20   mbarrene        }
   20   mbarrene
   20   mbarrene        /**
   20   mbarrene         * This function will modify the current CacheRecord for Data Information.
   20   mbarrene         * @param record
   20   mbarrene         */
```

```
  25  mbarrene        private boolean parseDataEntry(CacheRecord record, byte bytes [], int index){
  20  mbarrene
  25  mbarrene                //Grab the 32 bits that represent the metadata or data entry for this 128-bit record.
  20  mbarrene                Integer byteInt = ByteBuffer.wrap(bytes, index, 4).getInt();
  20  mbarrene                String hexString = Integer.toHexString(byteInt);
  20  mbarrene                String binaryString = Integer.toBinaryString(byteInt);
  20  mbarrene
  23  mbarrene                record.setDataByteInteger(byteInt);
  20  mbarrene                record.setDataHexString(hexString);
  20  mbarrene                record.setDataBinaryString(binaryString);
  20  mbarrene
  20  mbarrene                //Error Checking
  20  mbarrene                if(binaryString.length() != 32 || hexString.length() != 8){
  25  mbarrene                        //System.out.println("Record #" + this.recordCount_ + " is inconsistent with Hex and Binary data (Data Entry). Skipping...");
  20  mbarrene                        return false;
  20  mbarrene                }
  20  mbarrene
  20  mbarrene                //Trim the Binary String and determine the cache file pertaining to this 128-bit record.
  20  mbarrene                String temp = binaryString.substring(2,4);
  20  mbarrene                int cacheFile = Integer.parseInt(temp, 2);
  20  mbarrene
  20  mbarrene                CacheFile cf = null;
  20  mbarrene                switch(cacheFile){
  20  mbarrene
  20  mbarrene                case 0: cf = CacheFile.EXTERNAL; break;
  20  mbarrene                case 1: cf = CacheFile.CACHE_001; break;
  20  mbarrene                case 2: cf = CacheFile.CACHE_002; break;
  20  mbarrene                case 3: cf = CacheFile.CACHE_003; break;
  21  mbarrene                default: System.out.println("ERROR: Unrecognized Cache File Number: FFCacheParser.doParse().parseDataEntry()");
  23  mbarrene                return false;
  20  mbarrene                }
  20  mbarrene
  20  mbarrene                record.setDataCacheFile(cf);
  20  mbarrene
  20  mbarrene                //Then, trim the Binary String and determine the number of cache blocks that this 128-bit record occupies.
  20  mbarrene                temp = binaryString.substring(6,8);
  20  mbarrene                int numCacheBlocks = 1 + Integer.parseInt(temp, 2);
  20  mbarrene                record.setDataBlockCount(numCacheBlocks);
  20  mbarrene
  20  mbarrene                //If not an external file, trim the Binary String and determine the cache block number of this 128-bit record.
  20  mbarrene                int cacheBlockNum = 0;
  20  mbarrene                if(cf != CacheFile.EXTERNAL){
  20  mbarrene
  20  mbarrene                        temp = binaryString.substring(8, 32);
  20  mbarrene                        cacheBlockNum = Integer.parseInt(temp, 2);
  20  mbarrene                        record.setDataBlockNumber(cacheBlockNum);
  22  mbarrene
  22  mbarrene                        //Finally, determine the offset in the cache file wherein the data and metadata for this 128-bit record resides.
  25  mbarrene                        long offset = cf.getCacheBlockSize() * cacheBlockNum + 4096;
  22  mbarrene                        record.setDataFileOffset(offset);
  22  mbarrene
  25  mbarrene                        if(offset > 0 && offset < cf.getCacheFileSize())
  22  mbarrene                                return true;
  22  mbarrene                        else{
  25  mbarrene                                //System.out.println("ERROR: Offset is greater than the size of the Cache File (Data Entry). Skipping...");
  22  mbarrene                                return false;
  22  mbarrene                        }
  22  mbarrene
  20  mbarrene                }
  20  mbarrene
  22   mbarrene                //External Files: No more work to be done; finish parsing.
  23   mbarrene                else
  21   mbarrene                        return true;
  23   mbarrene        }
  23   mbarrene
  23   mbarrene
  25   mbarrene        protected boolean writeDataToFile(CacheRecord record) throws IOException{
  23   mbarrene
  25   mbarrene                String filename = String.format("%03d", record.getCacheRecordId());
  25   mbarrene                File file = new File(filename);
  23   mbarrene                file.createNewFile();
  23   mbarrene
  23   mbarrene                if(file.exists() == false){
  23   mbarrene                        System.err.println("File Failed to Create.");
  23   mbarrene                        return false;
  21   mbarrene                }
  23   mbarrene
  23   mbarrene                FileWriter fw = new FileWriter(file);
  25   mbarrene                CacheFile datafile = record.getDataCacheFile();
  25   mbarrene                CacheFile metadatafile = record.getMetadataCacheFile();
  23   mbarrene
  23   mbarrene                /*********************** Writing to this File ****************************/
  23   mbarrene
  25   mbarrene                /* ******************** Data and Metadata Entry Parsing ******************/
  25   mbarrene
  23   mbarrene                fw.write("---------------------------------------- \n");
  23   mbarrene                fw.write("Record #: " + record.getCacheRecordId() + "\n");
  23   mbarrene
  25   mbarrene                fw.write("Data:\n");
  25   mbarrene                fw.write("\t Hex String: " + record.getDataHexString() + "\n");
  25   mbarrene                fw.write("\t Binary String: " + record.getDataBinaryString() + "\n");
  23   mbarrene
  25   mbarrene                if(datafile != null && record.getDataEntryParsed() == true){
  23   mbarrene
  25   mbarrene                        fw.write("\t Cache File: " + datafile.getCacheFileName() + "\n");
  25   mbarrene
  25   mbarrene                        if(datafile != CacheFile.EXTERNAL){
  25   mbarrene
  25   mbarrene                                fw.write("\t Number of Cache Blocks: " + record.getDataBlockCount() + " " +
  25   mbarrene                                                datafile.getCacheBlockSize() + "-byte Block(s)\n");
  25   mbarrene                                fw.write("\t Offset Cachce Block #: " + record.getDataBlockNumber() + "\n");
  25   mbarrene                                fw.write("\t Offset in Bytes: " + record.getDataFileOffset() + " bytes\n");
  25   mbarrene                        }
  23   mbarrene                }
  25   mbarrene
  25   mbarrene                fw.write("Metadata:\n");
  25   mbarrene                fw.write("\t Hex String: " + record.getMetadataHexString() + "\n");
  25   mbarrene                fw.write("\t Binary String: " + record.getMetadataBinaryString() + "\n");
  23   mbarrene
  25   mbarrene                if(metadatafile != null && record.getMetadataEntryParsed() == true){
  23   mbarrene
  25   mbarrene                        fw.write("\t Cache File: " + metadatafile.getCacheFileName() + "\n");
  23   mbarrene
  25   mbarrene                        if(metadatafile != CacheFile.EXTERNAL){
  25   mbarrene
  25   mbarrene                                fw.write("\t Number of Cache Blocks: " + record.getMetadataBlockCount() +
" " +
  25   mbarrene                                                metadatafile.getCacheBlockSize() + "-byte Block(s)\n");
  25   mbarrene
  25   mbarrene                                fw.write("\t Offset Cache Block #: " + record.getMetadataBlockNumber() +
"\n");
  25   mbarrene                                fw.write("\t Offset in Bytes: " + record.getMetadataFileOffset() + "
bytes\n\n");
  25   mbarrene                        }
  23   mbarrene                }
  25   mbarrene
```

```
   23   mbarrene
   25   mbarrene                        /****************** Data and Metadata Header Parsing **************************/
   23   mbarrene
   25   mbarrene                        if(metadatafile != null && metadatafile != CacheFile.EXTERNAL && record.getMetadataParsed() ==
true){
   25   mbarrene
   25   mbarrene                                fw.write("Stored Data Header: \n");
   25   mbarrene                                fw.write(record.getData() + "\n");
   25   mbarrene                                fw.write("\nMetadata:\n");
   25   mbarrene                                fw.write("\t Fetch Count: " + record.getMetadataFetchCount() + "\n");
   25   mbarrene                                fw.write("\t Last Fetch: " + record.getMetadataLastFetched() + "\n");
   25   mbarrene                                fw.write("\t Last Modified: " + record.getMetadataLastModified() + "\n");
   25   mbarrene                                fw.write("\t Expiration Time: " + record.getMetadataExpirationTime() + "\n");
   25   mbarrene                                fw.write("\t Data Size: " + record.getDataObjectSize() + " Byte(s)\n");
   25   mbarrene                                fw.write("\t URL Size: " + record.getMetadataUrlSize() + " Byte(s)\n");
   25   mbarrene                                fw.write("\t Metadata Size: " + record.getMetadataSize() + " Byte(s)\n");
   25   mbarrene                                fw.write("\t URL: " + record.getMetadataUrl() + "\n\n");
   25   mbarrene                                fw.write("Metadata (From Server): \n");
   25   mbarrene                                fw.write(record.getMetadata() + "\n");
   25   mbarrene
   25   mbarrene                                fw.write("Remaining Slack Data: \n");
   25   mbarrene                                fw.write(record.getMetadataSlack() + "\n");
   25   mbarrene                        }
   25   mbarrene                        else
   25   mbarrene                                fw.write("\n\n\t Record # " + record.getCacheRecordId() + " has inconsistent data and
could not be fully parsed.\n");
   25   mbarrene
   25   mbarrene                        fw.close();
   23   mbarrene                        file.renameTo(new File(this.getOutDirectory(), file.getName()));
   23   mbarrene                        return true;
   20   mbarrene                }
   20   mbarrene
   20   mbarrene
   19   mbarrene                /**
   18   mbarrene                 * checkArgs():
   18   mbarrene                 *
   19   mbarrene                 * This function accepts the input program argument(s) and determines the
   19   mbarrene                 * directory path where the Firefox cache files live. If successful, the
   19   mbarrene                 * program will be ready for parsing these cache files.
   18   mbarrene                 *
   18   mbarrene                 * @param args
   18   mbarrene                 * @return
   18   mbarrene                 */
   19   mbarrene                public boolean checkArgs(String args[]) {
   14   mbarrene
   16   mbarrene                        try{
   16   mbarrene
   19   mbarrene                                // Directory we're looking for
   16   mbarrene                                File directory = new File(args[0]);
   18   mbarrene
   19   mbarrene                                if (directory.isDirectory() == true) {
   19   mbarrene
   18   mbarrene                                        System.out.println("Found the directory path: " +
directory.getAbsolutePath());
   20   mbarrene
   21   mbarrene                                        File cachefile = new File(directory.getAbsolutePath() + "/" +
CacheFile.CACHE_MAP.getCacheFileName());
   20   mbarrene                                        this.setCacheMap(CacheFile.CACHE_MAP);
   21   mbarrene                                        this.getCacheMap().setCacheFileObject(cachefile);
   21   mbarrene                                        this.getCacheMap().setCacheFileSize(cachefile.length());
   20   mbarrene
   21   mbarrene                                        cachefile = new File(directory.getAbsolutePath() + "/"+
CacheFile.CACHE_001.getCacheFileName());
   20   mbarrene                                        this.setCache001(CacheFile.CACHE_001);
   21   mbarrene                                        this.getCache001().setCacheFileObject(cachefile);
   21   mbarrene                                        this.getCache001().setCacheFileSize(cachefile.length());
   20   mbarrene
   21   mbarrene                                              cachefile = new File(directory.getAbsolutePath() + "/"+
CacheFile.CACHE_002.getCacheFileName());
   20   mbarrene                                              this.setCache002(CacheFile.CACHE_002);
   21   mbarrene                                              this.getCache002().setCacheFileObject(cachefile);
   21   mbarrene                                              this.getCache002().setCacheFileSize(cachefile.length());
   20   mbarrene
   21   mbarrene                                              cachefile = new File(directory.getAbsolutePath() + "/"+
CacheFile.CACHE_003.getCacheFileName());
   20   mbarrene                                              this.setCache003(CacheFile.CACHE_003);
   21   mbarrene                                              this.getCache003().setCacheFileObject(cachefile);
   21   mbarrene                                              this.getCache003().setCacheFileSize(cachefile.length());
   19   mbarrene
   20   mbarrene                                              if (this.getCache001().getCacheFileObject().isFile() &&
this.getCache002().getCacheFileObject().isFile() && this.getCache003().getCacheFileObject().isFile())
   18   mbarrene                                                          System.out.println("Discovered Important Cache Files in Directory
Path...Ready for Parsing.");
   18   mbarrene                                              else
   19   mbarrene                                                          throw new FileNotFoundException("ERROR: Unable to find
important Firefox Cache Files in Directory Path.");
   16   mbarrene                                  }
   19   mbarrene                                  else
   19   mbarrene                                              throw new FileNotFoundException("ERROR: Argument is not a Directory.");
   16   mbarrene
   19   mbarrene                      } catch (FileNotFoundException ex){
   19   mbarrene                                  ex.printStackTrace();
   19   mbarrene                                  return false;
   16   mbarrene                      }
   19   mbarrene
   14   mbarrene                      return true;
   14   mbarrene          }
   19   mbarrene
   12   mbarrene
   19   mbarrene
   19   mbarrene
   12   mbarrene }
mbarrene@elnux7.cs.umass.edu's password:
   22   mbarrene package hw1;
   22   mbarrene
   22   mbarrene import java.io.File;
   22   mbarrene import java.io.FileInputStream;
   22   mbarrene import java.io.FileNotFoundException;
   22   mbarrene import java.io.IOException;
   22   mbarrene
   22   mbarrene /**
   22   mbarrene  * Mario Barrenechea CMPSCI 365: Digital Forensics
   22   mbarrene  *
   22   mbarrene  * Hexdump:
   22   mbarrene  *
   22   mbarrene  * This program mirrors the UNIX command "hexdump" which takes in a file and
   22   mbarrene  * dumps its hex representation on the screen. Additionally, the --string
   22   mbarrene  * argument can be added to provide similar output as the UNIX command
   22   mbarrene  * "strings".
   22   mbarrene  *
   22   mbarrene  * First, a HexDump Object is created to maintain the command line arguments, and
   22   mbarrene  * then the proper function is called by determining whether the arguments call
   22   mbarrene  * for a string or a hex dump.
   22   mbarrene  *
   22   mbarrene  * @author mbarrenecheajr
   22   mbarrene  */
   22   mbarrene
   22   mbarrene public class Hexdump {
   22   mbarrene
   22   mbarrene          // Members
   22   mbarrene          private String m_filename = null;
   22   mbarrene          private boolean m_stringOutput = false;
```

```
22  mbarrene
22  mbarrene    // Getters
22  mbarrene    public String getFilename() {return this.m_filename;}
22  mbarrene    public boolean getStringOutput() {return this.m_stringOutput;}
22  mbarrene
22  mbarrene    // Setters
22  mbarrene    public void setFilename(String fn) {this.m_filename = fn;}
22  mbarrene    public void setStringOutput(boolean so) {this.m_stringOutput = so;}
22  mbarrene
22  mbarrene
22  mbarrene    /*** MAIN METHOD ****/
22  mbarrene    public static void main(String args[]) {
22  mbarrene
22  mbarrene        Hexdump hd = new Hexdump();
22  mbarrene
22  mbarrene        // Determine correctness of program argument(s).
22  mbarrene        if (hd.checkArgs(args) == true) {
22  mbarrene
22  mbarrene            //Either a HexDump or a StringDump
22  mbarrene            if (hd.getStringOutput() == true)
22  mbarrene                Hexdump.dumpStrings(hd.getFilename());
22  mbarrene            else
22  mbarrene                Hexdump.dumpHex(hd.getFilename());
22  mbarrene        }
22  mbarrene    }
22  mbarrene
22  mbarrene
22  mbarrene    /**
22  mbarrene     * dump()
22  mbarrene     * <p>
22  mbarrene     * This function parses the file with the filename parameter and correctly
22  mbarrene     * prints out three columns: 1) The number of bytes starting from the
22  mbarrene     * beginning of the file in hex, 2) the byte-for-byte hex representation in
22  mbarrene     * the file, and 3) the actual ASCII representation of each character (or
22  mbarrene     * '.' if NA).
22  mbarrene     */
23  mbarrene    protected static String dumpHex(String filename, int limit, int byteInterval, long skipBytes){
22  mbarrene
22  mbarrene        File file = new File(filename);
23  mbarrene        String result = "";
22  mbarrene
22  mbarrene        try{
22  mbarrene
22  mbarrene            FileInputStream fs = new FileInputStream(file);
22  mbarrene            int count = 0;
22  mbarrene            if(skipBytes != 0)
22  mbarrene                fs.skip(skipBytes);
22  mbarrene
22  mbarrene            while(fs.available() > 0){
22  mbarrene
22  mbarrene                if(count >= limit)
22  mbarrene                    break;
22  mbarrene
22  mbarrene                byte nextBytes [] = new byte [byteInterval];
22  mbarrene                fs.read(nextBytes, 0, byteInterval);
22  mbarrene
22  mbarrene                //Get the length of the bytes array (tells us where we currently are in the file)
22  mbarrene                count += nextBytes.length;
22  mbarrene
22  mbarrene                //Column #1: Count of Bytes in Hex
23  mbarrene                //System.out.printf("%04X:  ", count);
23  mbarrene                result += String.format("%04X:  ", count);
22  mbarrene
23  mbarrene
22  mbarrene                //Column #2: Print Hex Values of (ByteInterval) Bytes Each
22  mbarrene                for(int i = 0; i < nextBytes.length; i++){
```

```
22  mbarrene
22  mbarrene                        if(i == nextBytes.length/2 - 1)
23  mbarrene                            //System.out.printf("%02x  ", nextBytes[i]);
23  mbarrene                            result += String.format("%02x  ", nextBytes[i]);
22  mbarrene                        else
23  mbarrene                            //System.out.printf("%02x ", nextBytes[i]);
23  mbarrene                            result += String.format("%02x ", nextBytes[i]);
22  mbarrene                    }
22  mbarrene
22  mbarrene                    //Column #3: Print ASCII Characters
23  mbarrene                    //System.out.print("  |");
23  mbarrene                    result += "  |";
23  mbarrene
22  mbarrene                    for(byte b : nextBytes){
22  mbarrene
22  mbarrene                        if(b >= 32 && b < 127)
23  mbarrene                            //System.out.printf("%c",(char)b);
23  mbarrene                            result += String.format("%c", (char)b);
22  mbarrene                        else
23  mbarrene                            //System.out.printf("%c", '.');
23  mbarrene                            result += String.format("%c", '.');
22  mbarrene                    }
23  mbarrene                    //System.out.println("|");
23  mbarrene                    result += "|\n";
22  mbarrene
22  mbarrene                }//end loop
22  mbarrene
22  mbarrene                fs.close();
22  mbarrene
22  mbarrene            }catch (FileNotFoundException e){
22  mbarrene                e.printStackTrace();
22  mbarrene
22  mbarrene            }catch (IOException e){
22  mbarrene                e.printStackTrace();
22  mbarrene            }
23  mbarrene
23  mbarrene            return result;
22  mbarrene        }
22  mbarrene
22  mbarrene        /**
22  mbarrene         * dump()
22  mbarrene         * <p>
22  mbarrene         * This function parses the file with the filename parameter and correctly
22  mbarrene         * prints out three columns: 1) The number of bytes starting from the
22  mbarrene         * beginning of the file in hex, 2) the byte-for-byte hex representation in
22  mbarrene         * the file, and 3) the actual ASCII representation of each character (or
22  mbarrene         * '.' if NA).
22  mbarrene         */
22  mbarrene        private static void dumpHex(String filename){
22  mbarrene
22  mbarrene            File file = new File(filename);
22  mbarrene            try{
22  mbarrene
22  mbarrene                FileInputStream fs = new FileInputStream(file);
22  mbarrene                int count = 0;
22  mbarrene
22  mbarrene                while(fs.available() > 0){
22  mbarrene
22  mbarrene                    byte nextBytes [] = new byte [16];
22  mbarrene                    fs.read(nextBytes, 0, 16);
22  mbarrene
22  mbarrene                    //Get the length of the bytes array (tells us where we currently are in the file)
22  mbarrene                    count += nextBytes.length;
22  mbarrene
22  mbarrene                    //Column #1: Count of Bytes in Hex
22  mbarrene                    System.out.printf("%04X:  ", count);
```

```
22  mbarrene
22  mbarrene                    //Column #2: Print Hex Values of 16 Bytes Each
22  mbarrene                    for(int i = 0; i < nextBytes.length; i++){
22  mbarrene
22  mbarrene                        if(i == 7)
22  mbarrene                            System.out.printf("%02x  ", nextBytes[i]);
22  mbarrene                        else
22  mbarrene                            System.out.printf("%02x ", nextBytes[i]);
22  mbarrene                    }
22  mbarrene
22  mbarrene                    //Column #3: Print ASCII Characters
22  mbarrene                    System.out.print("  |");
22  mbarrene                    for(byte b : nextBytes){
22  mbarrene
22  mbarrene                        if(b >= 32 && b < 127)
22  mbarrene                            System.out.printf("%c",(char)b);
22  mbarrene                        else
22  mbarrene                            System.out.printf("%c", '.');
22  mbarrene                    }
22  mbarrene                    System.out.println("|");
22  mbarrene
22  mbarrene                }//end loop
22  mbarrene
22  mbarrene                fs.close();
22  mbarrene
22  mbarrene            }catch (FileNotFoundException e){
22  mbarrene                e.printStackTrace();
22  mbarrene
22  mbarrene            }catch (IOException e){
22  mbarrene                e.printStackTrace();
22  mbarrene            }
22  mbarrene        }
23  mbarrene
23  mbarrene
23  mbarrene
23  mbarrene        /**dumpStrings();
23  mbarrene         * <p>
23  mbarrene         * This function iterates through the hex dump, finds big-enough strings
23  mbarrene         * of characters that exist, and then prints them.
23  mbarrene         */
23  mbarrene        protected static String dumpStrings(String filename, int limit, int byteInterval, long skipBytes){
23  mbarrene
23  mbarrene            File file = new File(filename);
23  mbarrene            String result = "";
23  mbarrene
23  mbarrene            try{
23  mbarrene
23  mbarrene                FileInputStream fs = new FileInputStream(file);
23  mbarrene                int count = 0;
23  mbarrene                if(skipBytes != 0)
23  mbarrene                    fs.skip(skipBytes);
23  mbarrene
23  mbarrene                //Keep Track of Bytes pieced together to form strings.
23  mbarrene                String byteString = "";
23  mbarrene
23  mbarrene                while(fs.available() > 0){
23  mbarrene
23  mbarrene                    if(count >= limit)
23  mbarrene                        break;
23  mbarrene
23  mbarrene                    byte nextBytes [] = new byte [byteInterval];
23  mbarrene                    fs.read(nextBytes, 0, byteInterval);
23  mbarrene                    count += byteInterval;
23  mbarrene
23  mbarrene
23  mbarrene                    /*** Iterate through the bytes and determine unicode characters
```

```
23  mbarrene                  and consecutive characters that will make up a proper string ***/
23  mbarrene
23  mbarrene                  int unicodeBytePair = 0;
23  mbarrene                  for(int i = 0; i < nextBytes.length; i++){
23  mbarrene
23  mbarrene                      byte b = nextBytes[i];
23  mbarrene
23  mbarrene                      if(b >= 32 && b < 127){
23  mbarrene
23  mbarrene                          unicodeBytePair = 1;
23  mbarrene                          byteString += (char)b;
23  mbarrene
23  mbarrene                      }else if(b == 00 || b == 0){
23  mbarrene
23  mbarrene                          //Reset the Counter to Ignore the
23  mbarrene                          //second Byte in a Unicode Character
23  mbarrene                          if(unicodeBytePair == 1)
23  mbarrene                              unicodeBytePair = 0;
23  mbarrene                          else
23  mbarrene                              byteString = "";
23  mbarrene
23  mbarrene                      }else if (byteString.length() >= 4){
23  mbarrene
23  mbarrene                          //System.out.println(byteString);
23  mbarrene                          result += byteString + "\n";
23  mbarrene                          byteString = "";
23  mbarrene                          unicodeBytePair = 0;
23  mbarrene                      }
23  mbarrene
23  mbarrene                  }//end for loop
23  mbarrene                  }//end loop
23  mbarrene
23  mbarrene              fs.close();
23  mbarrene
23  mbarrene          }catch (FileNotFoundException e){
23  mbarrene              e.printStackTrace();
23  mbarrene
23  mbarrene          }catch (IOException e){
23  mbarrene              e.printStackTrace();
23  mbarrene          }
23  mbarrene          return result;
23  mbarrene      }
22  mbarrene
22  mbarrene      /**dumpStrings();
22  mbarrene       * <p>
22  mbarrene       * This function iterates through the hex dump, finds big-enough strings
22  mbarrene       * of characters that exist, and then prints them.
22  mbarrene       */
22  mbarrene      private static void dumpStrings(String filename){
22  mbarrene
22  mbarrene          File file = new File(filename);
22  mbarrene          try{
22  mbarrene
22  mbarrene              FileInputStream fs = new FileInputStream(file);
22  mbarrene
22  mbarrene              //Keep Track of Bytes pieced together to form strings.
22  mbarrene              String byteString = "";
22  mbarrene
22  mbarrene              while(fs.available() > 0){
22  mbarrene
22  mbarrene                  byte nextBytes [] = new byte [64];
22  mbarrene                  fs.read(nextBytes, 0, 64);
22  mbarrene
22  mbarrene                  /*** Iterate through the bytes and determine unicode characters
22  mbarrene                  and consecutive characters that will make up a proper string ***/
22  mbarrene
```

```
22  mbarrene                            int unicodeBytePair = 0;
22  mbarrene                            for(int i = 0; i < nextBytes.length; i++){
22  mbarrene
22  mbarrene                                byte b = nextBytes[i];
22  mbarrene
22  mbarrene                                if(b >= 32 && b < 127){
22  mbarrene
22  mbarrene                                    unicodeBytePair = 1;
22  mbarrene                                    byteString += (char)b;
22  mbarrene
22  mbarrene                                }else if(b == 00 || b == 0){
22  mbarrene
22  mbarrene                                    //Reset the Counter to Ignore the
22  mbarrene                                    //second Byte in a Unicode Character
22  mbarrene                                    if(unicodeBytePair == 1)
22  mbarrene                                        unicodeBytePair = 0;
22  mbarrene                                    else
22  mbarrene                                        byteString = "";
22  mbarrene
22  mbarrene                                }else if (byteString.length() >= 4){
22  mbarrene
22  mbarrene                                    System.out.println(byteString);
22  mbarrene                                    byteString = "";
22  mbarrene                                    unicodeBytePair = 0;
22  mbarrene                                }
22  mbarrene
22  mbarrene                            }//end for loop
22  mbarrene                        }//end loop
22  mbarrene
23  mbarrene
23  mbarrene                        fs.close();
23  mbarrene
22  mbarrene                }catch (FileNotFoundException e){
22  mbarrene                        e.printStackTrace();
22  mbarrene
22  mbarrene                }catch (IOException e){
22  mbarrene                        e.printStackTrace();
22  mbarrene                }
22  mbarrene        }
22  mbarrene
22  mbarrene
22  mbarrene        /**
22  mbarrene         * checkArgs()
22  mbarrene         * <p>
22  mbarrene         * This method is for parsing command line arguments and properly feeding
22  mbarrene         * them to the Hexdump object so that the hex representation is sent as
22  mbarrene         * output (and maybe the strings output).
22  mbarrene         *
22  mbarrene         * @param args
22  mbarrene         * @return
22  mbarrene         */
22  mbarrene        private boolean checkArgs(String args[]) {
22  mbarrene
22  mbarrene                boolean done = false;
22  mbarrene                String filename;
22  mbarrene
22  mbarrene                // For each argument coming in...
22  mbarrene                for (int i = 0; i < args.length; i++) {
22  mbarrene
22  mbarrene                        // If this String is the file flag and there is still
22  mbarrene                        // an argument to parse (hopefully the filename)...
22  mbarrene                        if (args[i].equals("--file") && i < args.length - 1) {
22  mbarrene
22  mbarrene                            filename = args[i + 1];
22  mbarrene
22  mbarrene                            File file = new File(filename);
```

```
22  mbarrene                    if (file.exists() == false)
22  mbarrene                        return false;
22  mbarrene
22  mbarrene                    this.setFilename(filename);
22  mbarrene                    done = true;
22  mbarrene                }
22  mbarrene
22  mbarrene                else if (args[i].equals("--strings"))
22  mbarrene                    this.setStringOutput(true);
22  mbarrene
22  mbarrene            }// end loop
22  mbarrene
22  mbarrene            if (done == true)
22  mbarrene                return true;
22  mbarrene            else
22  mbarrene                return false;
22  mbarrene        }
22  mbarrene
22  mbarrene }// end HexDump class
```