Digital Forensics - Using Perl to Harvest Hash Sets

Douglas White & John Tebbutt
June 2004
Software Diagnostics and Conformance Testing Division
National Institute of Standards and Technology
Email: nsrl@nist.gov

Introduction

The National Software Reference Library (NSRL) is designed to collect software from various sources and incorporate file profiles computed from this software into a Reference Data Set (RDS) of information. The RDS is used by law enforcement, government, and industry organizations to review files on computers by matching file profiles in the RDS. This helps, for example, to efficiently determine which files are important as evidence on computers or file systems that have been seized as part of criminal investigations [1].

When software destined for the National Software Reference Library arrives in our library room, it is labeled and classified according to information such as application name, version, manufacturer, etc., and unique identifiers are allocated to the software package and its constituent media. At this point, the extraction of the package's files can occur in a process known as batching.

The batching process takes files from the original media and places them on our file server. Once batched, file profiles are constructed for every file on the media, using a loose cluster of computing nodes . These profiles are comprised of information that allow unique identification of every file. Part of the profile information is a set of hash values [2] [3] that are computed, condensed representations of individual files [4] [5]. The hashes are the most popularly used features of the NSRL RDS.

This paper focuses on the use of Perl to automate the harvesting of hash values from software package media in a way which ensures the highest degree of correlation between the set of files contained on the media and the set of files which is transferred onto working systems during the package installation process.

We believe that the method we describe for hashing a library of files is sufficiently generic that it may be used, with only minor modifications, in a wide range of applications. For example, code unraveling, decryption, signal processing, etc. Law enforcement, digital archival, corporate security and forensic investigation applications can also be developed within this framework.

Components

For many reasons, the NSRL project is migrating to an open source environment, specifically using Linux, Apache, MySQL and Perl (LAMP) for the core technologies. To this end, our goal is to develop Perl code that can be used by computing nodes, running various operating systems (e.g. Windows, Linux, Mac OS), using common modules available from the Comprehensive Perl Archive Network (CPAN - www.cpan.org). We have accomplished this for the actual calculation of the hash values; Digest::SHA version 4.3.1, Digest::MD5 version 2.33, Digest::MD4 version 1.3 and String::CRC32 version 1.2 have proven that they provide the expected results

However, the computing nodes must both hash every file encountered **and** hash the contents of any "archive" type of file (e.g. zip file, tar file, cab file, uuencoded file). The diversity of the operating systems and possible archive file formats means that we need to provide an interface to allow third-party applications to process the archive files unless and until Perl modules exist to handle these formats and can be tested.

Custom Modules

In order to achieve our goal of extracting all files from distribution media, hashing those files and storing the resulting hashes, along with relevant identification information, etc., we built the following modules:

- **NSRL::Magic** - identify and classify archive files;
- **NSRL::Unarc** - extract files from archives;
- **NSRL::Config** - load configuration information for seamless, platform-neutral operation;
- **NSRL::Hash** - hash the contents of files and strings;
- **NSRL::DB -** provide database interface for storage of hash set information.

Identifying Files to Act Upon

Our research has shown that, when software is installed onto a computer, some 60% - 80% of the installed files are identical to files contained on the installation media (CDROMs, DVDROMs, downloads, floppy disks, etc.). However, the installed files often are not merely copied from the media: many files are contained in archives, and are extracted and placed onto the target machine during the installation process. Thus it is not sufficient to merely hash every file on the distribution media: we also need to identify any archives and extract and hash the files they contain. In a sense, we need to mimic the installation process in order to get the maximum information from the distribution media.

During the processing of distribution media files, archive files are identified using a specialization of the familiar magic number technique. Once identified as archives, their contents can be extracted into a "sandbox" for subsequent hashing. Identification is through the Perl module NSRL::Magic, which uses the CPAN module File::MMagic in combination with a modified Unix-type magic number file.

Adapting the Magic File

We chose File::MMagic to do magic file processing because it allows the specification of an external magic file, which enables us to use our own, marked-up magic file. Specifically, we took a generic *nix magic file (e.g. /etc/magic on a Linux system) and appended NSRL-specific tags to the lines describing archive files. For example:

```
0  string  MSCF\0\0\0\0   Microsoft cabinet file data archive, [NSRL|CAB]

0  string  PK\003\004     Zip archive data [NSRL|ZIP]

0  string  \037\213       gzip compressed data archive [NSRL|GZ]
```

The tags are inserted into the magic file to simplify the automation of file type identification; they are appended to existing text as opposed to replacing it for diagnostic and readability purposes. Currently we use magic numbers only to identify archive files, so we also remove all non-archive entries from our custom magic file. This has the advantage of speeding up processing considerably, as we are able to trim a file of some 9,000+ lines to less than 500 lines.

NSRL::Magic uses File::MMagic to determine the type of a file, based on our custom magic file. If File::MMagic can match a file, it returns the text string associated with the file type (e.g. "gzip compressed data archive [NSRL|GZ]"), otherwise the returned string is empty. NSRL::Magic parses the return string, looking for "[NSRL|" and simply returns whatever lies between it and the first subsequent "]". So, in the above example, "GZ" would be the return value. If the magic string has zero length, the special value "NARC" (Not an ARChive") is returned. If the string is non-zero in length but has no NSRL tag, the special value "UNK" (UNKnown file type) is returned. This latter functionality is included for robustness and for planned for future use, when the NSRL magic file is expected to contain entries for non-archive file types.

Identifying Applications to Act Upon a File

One of our primary aims in using Perl is to migrate away from a proprietary architecture in order to provide a set of tools which is usable across multiple platforms. This enhances our efforts in two important ways: it expands the range of software we are able to catalog for the NSRL; and it broadens the applicability of key components of the software for general release.

However, our "pure Perl" philosophy comes up short when it comes to archiving/extraction utilities: while CPAN contains many modules for archiving and extraction, most appear to be wrappers for *nix libraries, and we could find no support for some important archive types (notably Microsoft cabinet files and various Apple formats).

Our solution was to combine a generic archive extraction module with a configuration file containing platform-neutral and platform-specific information in a well defined format. Specifically, for each supported archive type, if a pure Perl module exists to handle the type, it will be used, regardless of platform. Otherwise, platform-native software is called to handle the extraction.

Platform Configuration

The NSRL::Config module expects a configuration file in the .ini style. Comments are entire lines that begin with pound (#) or semicolon (;) and blank lines are acceptable.

Config::Tiny version 1.6 supplies the core of the module, and there is a "root" or global section of the configuration file which applies to all platforms, followed by zero or more sections with platform-specific support application information. The following is an example of a configuration file that supports four operating systems: generic Linux, Windows 98, Windows 2000 and Mac OS X.

```
# example default NSRL configuration file

# the key=value mechanism here supports case sensitivity.


# root section
```

```
magic=true

recurse=true


[linux]

recurse_types=TAR,GZ,UU,DD,ISO

TAR=/sbin/tar -xvf <$SRCFILE$>

GZ=/bin/gunzip -c <$SRCFILE$> > <$DESTDIR$> <$DESTFILE$>

UU=uudecode <$SRCFILE$>

DD=mount -t loopback -o ro <$DESTDIR$>  <$SRCFILE$>

ISO=mount -t iso9660 -o ro <$DESTDIR$>  <$SRCFILE$>


[win98]

recurse_types=TAR,ZIP,DOS

TAR=tar -xvf <$SRCFILE$>

ZIP=winzipc -d <$SRCFILE$>

DOS=extract <$SRCFILE$> <$DESTDIR$> <$DESTFILE$>


[win2000]

recurse_types=CAB

CAB=cabarc -x <$SRCFILE$>


[macosx]

recurse_types=DMG,SIT

DMG=mount <$SRCFILE$>

SIT=unstuff -R <$SRCFILE$>

# limit platform to these types from perl %ENV

HOSTTYPE=powermac

MACHTYPE=powerpc

OSTYPE=darwin
```

Parsing the Configuration File

The root section of the configuration file holds information critical to any instance of the NSRL code. This includes specifying the use of the magic file information and whether recursion into archive files is to be done,

as the example above shows. There are other settings which pertain to the number of processors in the computer, timeout periods to prevent runaway processes, database interface information, etc.

An array is created which stores the platform section data; for the example above, this array would hold the four operating system values ("linux","win98","win2000","macosx"). If the NSRL code is run on a platform that does not fall into these categories, it will not have any platform-specific application support to manipulate archive files.

An array is created which stores the archive types supported by the platform on which the code is running; for the example above, if the code is run on a Linux platform, this array would hold the values("TAR","GZ","UU","DD","ISO"). The values in this array correspond to the NSRL-specific tags from the magic file, described above. Only these archive types can be processed on this platform.

Finally, there is a subroutine which recalls the generic support application string from the configuration information, based on the operating system and the archive type. The result of this subroutine is the *sysCall* parameter used in a call to NSRL::Unarc::expand.

By this method, we can specify the operating systems on which the NSRL code may perform recursive archive file hashing, the types of archive files that can be processed on each operating system, and the structure of the system call for each supporting application.

Performing an Action Upon a File

The NSRL::Unarc module is used to extract the contents of archive files. It exposes a single subroutine, expand(), the parameters of which are as follows:

sysCall - a string containing the platform-appropriate system command to extract the contents of the archive. The string contains substitution fields which are replaced by the values of the parameters that follow. The value of each field is given in parentheses after the description of the corresponding parameter.

archive - the fully-qualified pathname of an archive file (<$SRCFILE$>)

sandbox - the fully-qualified pathname of the location in which the contents of the archive are to be placed (<$DESTDIR$>)

destfile - the pathname of the destination file, relative to the sandbox directory (required for some native decompression commands); or null if not required (<$DESTFILE$>). If destfile is null, there should be no reference to <$DESTFILE$> in the *sysCall* parameter, and vice versa.

File extraction is achieved through the composition of a platform-appropriate command, substituting the generic fields in sysCall with information on the archive file to be processed and invoking the command using backticks.

Code Metrics

In 25 tests of the code on a random sampling of common media encountered by the NSRL, this code was able to hash and manipulate 29,942 MB of data in 78,315 seconds of CPU time. The test data was comprised of 290,161 files, of which 45,073 (15.5%) were archive files. The operations were performed on a test configuration comprising three computers: an Intel-based 2GHz PC with 2GB RAM running Windows 2000, an Intel-based 2GHz PC with 2GB RAM running Linux, and an Apple dual 2GHz G5 with 8GB RAM running OS X.

This yields the effective rate of 382 KB/second for this code. Given the computations being performed and the overhead of the framework, this is surprisingly good. A simple method to increase this throughput is to add

more machines. The working NSRL cluster typically contains 12 computers in a mix of the classes described above, bumping the throughput up to 4.5 MB/second.

Other Uses for this Framework

We believe this framework is sufficiently generic to be applicable to a range of file manipulation tasks requiring the extraction and/or examination of file contents. The framework can be used as a wrapper for heterogenous tasks which need to be performed on multiple computers across different operating systems.

For example, if we have a collection of image files that may potentially contain data embedded with a steganography tool, we can instruct the cluster of working computers to apply steganographic prediction algorithms to each file. A brute force method of attempting to unsteg files with popular tools is another possibility. Combine this with available password cracking tools, and you have a formidable automated environment that can evaluate the potential of covert communication and attempt to recover the message or data.

The NSRL has begun investigating the use of signal processing algorithms with respect to image and audio files. While hashes are not the most efficient long-term method to uniquely identify the dynamic data in images and audio, it may be possible to use other algorithms to determine statistical probabilities of similarities. There are applications which can be used to find classes of images, such as landscapes, flowers, human figures. Likewise with the frequency distribution of audio. A cluster of computers could be tasked with identifying potentially illegal images or audio, despite simple manipulation of the original.

Another use for this framework is in the support and maintenance of digital archival material, such as that maintained by various libraries. We currently use it to track historic files on various media or media images. Additionally, the framework lends itself well to the support of historic file formats: as long as an operating system and/or file system can be recreated or emulated, the applications of that era can be automated to perform various functions, for example, file format conversion.

In the short history of the NSRL, we have seen the collected data and the application environment find use in fields that were not among the disciplines that we considered initially. This framework is involved in our work with law enforcement data collection and with our work on archival digital information, and we hope to find other domains in which it may be applied.

While the custom Perl modules described in this paper are not generally available at this time, the NSRL developers have built them to CPAN guidelines and hope to distribute them in future.

Disclaimer: Vendor and product names are used only to provide specific information about hardware, software and processes used, not as endorsements.

Bibliography

[1] Fisher, G. (2001) NIST ITL Bulletin, "Computer Forensics Guidance".
https://www.nsrl.nist.gov/itlbulletin.html

[2] Rivest, R.L. (1992) IETF RFC1321, "The MD5 Message Digest Algorithm".
http://www.ietf.org/rfc/rfc1321.txt

[3] U.S. Department of Commerce. (1995) NIST FIPS 180-1, "Secure Hash Standard".
https://www.itl.nist.gov/fipspubs/fip180-1.htm

[4] Boland, T. and Fisher, G. (2000) "Selection Of Hashing Algorithms".
https://www.nsrl.nist.gov/documents/hash-selection.doc

[5] Pieprzyk J. and Sadeghiyan, B. (1993) "Design of Hashing Algorithms". ISBN 0-387-57500-6