Custom Logfile Analysis Utilities & an eBay Image Theft Detector

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2003 as GuruGram #28
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

My **ISP** has been balking at paying the outrageous fees for a new **Webtrends** upgrade, so I thought I might work up some freebie code that does a faster and better job of what **Webtrends** was supposed to be doing in the first place.

Besides all the usual data extraction, my new and easily customizable routines let you quickly detect **eBay** site wide **image popularity** and **possible theft**. They also give you a detailed **visitor-by-visitor transcript** of who did what to you when.

You'll find this new routine as **LOGRPT01.PSL** and a typical demo sample printout can be found as **LOGRPT01.LOG**. A Sample logfile for testing can be found up as **EX031126.LOG**.

My **GONZO.PS** utilities are strongly recommended but not required for the current code version, while additional use details appear in our **PostScript Library**.

A few of the preliminary report features currently included are...

Total hits for session.
Total pageviews for session.
Hits per page viewed.
Pages viewed by popularity.
Total pages visited at least once.
Total pages visited.
Average visits per page.
Images downloaded by popularity:
Total images visited at least once.
Total images downloaded.
Average visits per image.
Total new ad banners delivered.
Files downloaded by popularity:
Files downloaded at least once.

Total files downloaded.
Average visits per file.
Utilities downloaded by popularity:
Utilities downloaded at least once.
Total utilities downloaded.
Average visits per utility.
Specific crucial file download stats.
404 errors by severity:
Unique 404 file not found errors.
Total 404 file not found errors.
404 file errors as a hit percentage.
eBay image requests by popularity:
Total eBay images per item requests.
Total eBay images requested.
Average eBay per item requests.
Filtered referrals by popularity:
Total unique and useful referrals.
Total useful referrals.
Average visits per Referral.
Filtered search queries by popularity.
Total search queries.
Total unique search queries .
Average search query repeats.
Visitor log array creation.
Individual visitor activity reports.

This, of course, is just backing up for a good start. Many thousands of other analysis features are easily added. Generally, anything available with **Webtrends** can be done faster and better in a more easily customizable format.

At present, the utilities generate simple text reports containing only what you want exactly the way you want it. Because of the power of **PostScript**, arbitrarily fancy graphics can be wrapped around the core routines for client presentation. As could auto scripting. Present utilities are for daily reports only for a **single url**. Files are easily written to for monthly or other extended analysis

In general, you first make sure your daily log file is **locally** available. You then take a short and standard ASCII textfile utility and bring it up in a word processor or editor. You modify your filenames and any details or particulars of reporting any way you care to, and then resave the utility **as a standard ASCII textfile**.

Your modified and saved textfile is then sent to **Acrobat Distiller**. A log report file is then generated, typically in well under two minutes for a fancier site. The logfile may then be viewed, interpreted, or used for further analysis.

What is in a logfile?

Your **ISP** Internet Service Provider keeps track of your website activity. By creating **logfiles**. These ordinary printable textfiles consist of an info header and one info line for every website hit. Typically several megs daily. Incredibly useful stats can be either directly or indirectly extracted from your web logs...

If your ISP does not make your logfiles available, then DEMAND that they do so! Or change to a new ISP.

Logfiles may be used at the ISP end to extract system wide stats, or may be downloaded to individual sites for your own personal or commercial website use. There are several different styles of log files, and different reports may have varying content...

It is extremely important that logfile reading software EXACTLY matches the logfile in use!

A typical logfile source might be from **Microsoft Internet Information Services 6.0**. This consists of four machine readable header lines that start with an **#** identifying delimiter. Each site hit then consists of **one** line of a number of fields, typically seventeen. Each field is delimited by a space, and no spaces are allowed within any particular field. Lack of data in a field is designated by a **(-)** hyphen.

Field sequence is shown, of all places, on the **#Fields:** comment line. In the case of my ISP's present **logfile formats**, there are seventeen fields presented in this specific order...

date	Ten byte text string in a **2003:07:12** year:month:day format.
time	Eight byte text string in a **15:32:42** hours:minutes:seconds format.
s-ip	The url of **your** site being served. Such as **24.120.195.24** The length may vary, delimiting on the decimal points.
cs-method	The requested action as one uppercase word. **GET** and **POST** are common.
cs-uri-stem	The requested file to be delivered, such as **/webwb01.asp**.

cs-uri-query	Additional delivery info requested, such as a search string. Often unused as **(-)**.
s-port	Server port over which info is delivered. Returns **(80)** for my ISP.
cs-username	Name of user making request. Returns **(-)** if anonymous or DNS lookup not provided.
cs-ip	The URL of **their** site being served. Such as **66.234.212.133**. The actual website name can sometimes be found by a DNS lookup by using **Whois**, but may be an ISP doing anonymous dialup or wireless sharing. Do note that several individual websites could share the same cs-ip if they happen to request at the same time.
cs(User Agent)	Webserving software and system hardware being used by site making request. Often **Mozilla** running on a **Windows** platform.
cs-(Referrer)	Name of the **previous** page visited by the user making the request. Shows you which page they were on when they asked for yours. Can be exceptionally valuable.
sc-status	Reports on the success of the transaction. Some typical **status results** are...

200	OK
201	Created
202	Accepted
203	Non-Authoritive
204	No Content
205	Reset Content
206	Partial Content
300	Redirected
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	Not Used
307	Temporary Redirect

400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	File Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Too Large
414	URI Too Long
415	Unsupported Media
416	Range Not Satisfied
417	Expectation Failed
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	Version Not Supported

sc-substatus	Additional status info. Usually **(0)**.
Win32status	Windows32 status info. Usually **(0)**.
sc-bytes	Bytes delivered **to** visitor.
cs-bytes	Bytes received **from** visitor.
time-taken	Response time in **milliseconds**.

The popular **Apache** logfile format may differ somewhat from this **IIS** format shown. Once again, **be sure to match the format and fields to your analysis utilities!**

The problem with just looking at a logfile is that they are utterly overwhelming. And any useful piece of info is scattered over many lines in obscure order. The trick is to properly **isolate**, **rearrange**, and then **present** logfile info of interest so it is easily viewed and interpreted.

One final caution before continuing …

Any end nulls MUST be removed before you do any further processing of a log file!

Extracting Logfile Data

You normally start by making a logfile locally available on your system. Typically by **FTP** from your ISP. While it is possible to continually read and re-read the disk based logfile to get specific data, it is usually ridiculously faster and far better to capture the data first to software arrays or other structures. These arrays are then quickly and easily analyzed for specific results.

My approach to most any technical problem is to use the superbly wonderful **PostScript** as a general purpose computer language. This is normally done by creating a small ordinary ASCII textfile and sending it to a host based **Acrobat Distiller**. Acrobat then generates useful output in the form of a PDF file, reporting activity log files, and (optionally) new disk based data files. Following the secrets in **DISTLANG.HTML**.

For this logfile analyzer, our main interest will be in the generated PDF activity logfile where all of our useful results will appear. Thus throwing away the baby and drinking the washwater. You can safely ignore the expected **WARNING: No PDF File Produced!** errors. Later on, fancy graphics can get added.

The web logfile data will first get isolated into seventeen **PostScript** arrays. Those arrays can then be read, sorted, and otherwise manipulated to generate our useful results. Normally a PS array has a **fixed** length. Too short and you get errors, and too long and you have problems with ending nulls. Thus, **all PostScript array lengths must ~exactly~ match their intended content.**

You can also use this sneaky trick to dynamically expand a PostScript array…

```
/myarray mark myarray aload pop newitem ] store
```

What happens here is that a new and longer array gets defined by **mark**ing a new array start, unloading the old array onto your stack, adding your new element, and completing the new array with a closing bracket. The length count goes up by one.

While this cute stunt will work great for most of our analysis, dynamic expansion may be way too slow for the half million or more actions needed for any initial extraction. Instead, we will make **two** passes through our disk based logfile. Both passes will ignore comment lines and very short lines.

You may want to have **LOGRPT01.PSL** on hand and up in an editor or word processor to help you understand what follows here.

The first pass will determine how many lines are needed for capture. And be used to set the size of seventeen arrays that have been named **datearray**, **timearray**, **s-iparray**, **cs-method**, etc... The second disk reading pass will completely fill the arrays with the actual data needed.

The filling process consists of reading one valid line at a time, and recursively nesting sixteen searches on the space **( )** delimiter. A **PostScript** search will return a **pre**, **found**, and **post** string. **Pre** is saved to the current array of interest, **found** is flushed with a **pop**, and **post** is recursively reused to pick up the next data string of interest. The last string **post** is "free" and is stuffed into the final array.

One subtle gotcha at this point: If you have a string you are reusing and put it in an array, PostScript will enter only a **pointer** to that string into your array. **If the string changes later, so does the array!** Leading to wildly wrong and utterly mystifying results. The solution is to **dereference** your strings **before** they go into your arrays...

```
. . . . dup length string cvs . . . .
```

This creates a new, unique, and "safe" string for private entry into your array. The initial string on the stack is free to be reused and redefined any way you like without hurting the inplace array data.

Our net result at this point is to be finished with any diskfile reading and to have seventeen arrays of useful logfile data internally available for further processing.

Analyzing the data

At this point, you can decide what website info you want in what format. In general, you create new arrays of arrays. Each sub array entry will be an array in itself. Eventually containing a data string, a popularity count, and possibly additional data elements.

Typically, you will read data of interest from your primary arrays and rewrite them into your secondaries. Something of interest will either **add to** the count of an existing secondary entry or else **create** a brand new secondary entry.

These secondary arrays will initially be unsorted, since you have no early way of knowing how many times something gets repeated. A plain old **bubble sort** can then be used to **rearrange** each secondary array into order of popularity.

Yeah, bubble has a laughingly bad rep, but it is simple, nonsubtle, and uses few resources. It seems more than fast enough to sort a few hundred items. And can easily be further improved.

Bubble works by comparing the counts in array **entry 1** against array **entry 2**. If **entry 2** is bigger, they are swapped. **2** is then compared against **3**. If **3** is bigger, they are swapped. The first pass will end up with one of the **lowest** counts at the bottom of the pile at the end of the array. You then repeat the process **n** times for an array of length **n**. Or thereabouts. The final result is an ordered sorting by popularity.

Useful info can then be extracted from your sorted secondary arrays. For instance, lists of files can be output in popularity order. The number of files can be extracted and reported by using the **length** operator. The total number of file hits can be extracted by viewing each popularity count and adding them up. The ratio of file visits per file is easily calculated by a simple division.

Some Specifics

Let's briefly review how some of the useful data gets generated by making up **ordered secondary arrays** and then extracting desired info from them.

Depending upon your website arrangement and content, some approximations and assumptions may be involved, and some results may end up a tad on the fuzzy side. Nonetheless, incredibly useful results can often be reported and analyzed.

Let's see. The length of your primary arrays immediately tells you the number of **website hits**. A list of **page views** can be generated by making some assumptions about what a page view is. On my website, most **.asp** files are page views. And those less than fifteen characters long will be primary page views that include **banner ad** delivery. My **.html** files can largely be ignored since they are usually a redirect or downloadable data.

So, I'll go thorough the **cs-uri-stem** array and search for the **.asp** files. And then create a **pageviews** secondary array. After sorting, they are output by using the PostScript **==** command. You also have the (eventual) option of writing this data to disk for further long term analysis.

The length of your **pageviews** secondary array tells you the number of **unique page views**, while the total of the individual hit counts tells you the **total number of page views**. Their ratio gives you the **views per page**.

Similarly, most of my images are in an **/images/bargs/** subdirectory from which an **imagehitlist** secondary array can be generated and similar data extracted. Many of my downloadable files appear in a **/glib/** subdirectory. For which your **filehitlist** can be generated and exported. The same goes for my utilities which are often found in a **/psutils/** subdirectory and used to create a **pshitlist** secondary ordered array and report.

The number of **banner ads delivered** can be found by once again searching for **/banners/** in the **cs-uri-stem**.

Sometimes, specific file activity might be of special interest. I particularly like to know how many **GONZO.PS** downloads I get daily. Custom download reports are easily gotten by once again reading a previously generated secondary array such as **pshitlist**. And then extracting a special count report.

Reporting **file not found** errors are only slightly more complex. First, you have to go through **sc-status** to pick off the **position** of each **404**. Then you use this position pointer to go into **cs-uri-stem** to find out what it was that made the bad request in the first place. These can be gathered into an **e404hl** array and then output for viewing.

You'll find at least three different types of **404 errors**. The first type are ones **you** caused by website boo-boos. These you should seek out and correct. You'll want to spend the most time on the ones that are causing the most grief.

The second type are fumble fingered typos from visitors that are unlikely to ever happen again. And the third type are malicious attempts at cracking your website for root access. These can usually be identified by a **../winmt/system32/cmd.exe** fragment or similar obviously bogus requests. We'll see an example shortly.

Referrals are enormously useful to find which **outside** sites are sending visitors your way, which **inside** paths are the most popular, what your **eBay** sales are up to and who is **searching** for what from where.

To generate a report of **useful referrals**, you create yet another **freflist** secondary array. By going into **cs-referrer** to pull out all referrals longer than five characters. An **referral exclude list** is then completely searched using **forall** to eliminate any unwanted referrals. My own exclude list currently consists of variations on my **tinaja** website name, the **(?)** indicating a search, and a few common referral mistakes. Then, your filtered visitor referrals get sorted and reported as usual.

Search queries work the opposite. You go through **cs-referrer** and seek out **only** those entries with a **(?)** in them. Optionally excluding any from **eBay**. Reports can get further fancified by noting the **specific** use of **(?)** by any particular engine. Such as **Google** starting its search query string with **q=**. Or **Yahoo** with **p=**.

If you are an eBay seller...

Custom outputs may be of interest to **eBay** users. You can easily make up a secondary array that contains only eBay item numbers, and your images they returned. For an ordered list of your current eBay item popularity. Such as ...

```
[(eBay #item=2576660841 /adepts01.jpg) 20]
[(eBay #item=2205626364 /nuke01.jpg) 17]
[(eBay #item=2576657732 /MT31001.jpg) 13]

                                    etc...
```

To generate this list, the **position** of each **eBay** hit is found and the **item request** gets extracted. The position pointer is used to reach into the **cs-uri-stem** array to extract the image gotten. The gathered strings are then shortened, adjusted, and merged (using **mergestr** from my **Gonzo Utilities**) to produce the above report messages.

The **eBay** search query string can have several entries and be in any order. You can view the sourcecode to find my current isolation algorithm in use.

You can instantly find out how well each and every one of your offered items is doing. All with a few mouse clicks. And not having to mess with individual **eBay** counters or repeated **eBay** site views. By using fairly obvious image names, **no lookup of item numbers is required**! You can optionally save your daily data to get long term totals of each offered item.

Further, if you are **not** currently offering an item that seems oddly popular, this may mean someone else has stolen your image. You also have the exact **eBay** offering number for the theft. And can file a **NODI** or replace your image with one of an appropriately clad individual aggressively pioneering new methods of animal husbandry.

More on these techniques on our **Auction Help** library page. Do note that an occasional nonlisted image hit or two can be expected from a valid eBay user searching on back items.

Creating Visitor Activity Reports

You can go well beyond **webtrends** by making **individual visitor activity reports**. These let you find out what each and every visitor was up to on your site one on one. This tells you the exact and total path the visitor took through your site, which activities they were up to on each page, and where and when they left.

Ferinstance, here is a "good guy" access example...

```
68.238.160.217 took 18 minutes and 34 seconds to visit...

     (/picup01.asp)
     (/muse01.asp)
                    (/glib/muse109.pdf)
                    (/glib/muse99.pdf)
     (/tinaja01.asp)
     (/blat01.html)
                    (/glib/dontsick.pdf)
                    (/glib/emergop4.pdf)
                    (/glib/ratholes.pdf)
                    (/glib/marcia.pdf)
```

Here we see that the visitor went to four page views and then downloaded six reference tutorial files. We also note they are a **PIC** microprocessor fan, since they did not enter via the main website **home page**.

And here is a modified fragment of a "bad guy" site piracy attempt…

```
62.XX.76.YYY took 0 minutes and 58 seconds to visit...

   (/Default.asp)
                    (/MSDDC/root.exe)
                    (/f/winmt/system32/cmd.exe)
                    (/PCServer/cwinnt/system32/cmd.exe)
                    (/Rpg/cwinnt/system32/cmd.exe)
                    (/scripts/..Q../winmt/system32/cmd.exe)

                                                  etc...
```

We see not one attempt at viewing a real page or downloading a legitimate file. Instead, a mix of blatant grabs at obtaining system control. Sending Bruno to the ISP of 62.XX.76.YYY for attitude relateralization might not work if the sending site has also been spoofed.

A modified three element **visitorrawdict** secondary array gets used for your user logs. One visitor might look like this…

```
[ (204.24.574.34) 8 [123 146 157 158 159 332 347 348 ] ]
```

Here the first subarray entry is the url of the visitor. The third entry is an array of website hit positions. These will typically be intermixed with other visitors and rarely will be sequential. The second entry is the number of hit counts equal to the length of the third array. The length was kept in the same position as the earlier arrays so the same bubble sort could be reused.

To present the hits in useful form, you may want to ignore bunches of them. I am mainly interested in the major web page visits and their library downloads. So, hits involving banners or GIF's are discarded. As are design elements. A two column presentation is done with **.asp** or **.html** files of 15 characters or less in the first column and (mostly) downloads in the second. Repeat hits are rejected, since these are most likely a PDF byte range retrieval or a delivery glitch.

The visit time is approximated by finding the **earliest** and **latest** hits on the **time-taken** primary array. These are read as ASCII characters, converted to numeric seconds, and subtracted. Times are approximate since there could have been two or more sessions. And since the visitor is likely to spend additional time viewing their last page or download.

A typical visitor will also use their back arrow to view already downloaded pages. These are cached on the visitors machine and will not show up in these logs. But they sometimes can be inferred.

A curious gotcha: On my machine, visitor log creation slowed down dramatically above 9000 hits or so. This seems somehow related to an apparent PostScript garbage collection bug. My temporary workaround was to initially turn garbage collection off and then collect **once** every 2500 hits. This made the problem vanish on my machine but may not on yours. Please **report** any and all difficulties in this area.

Note that a **-2 vmreclaim** turns garbage collection off, a **1 vmreclaim** does **one** immediate garbage collection, and a **0 vmreclaim** restores automatic garbage collection operation.

Individual user log reports do raise issues involving abuse and invasion of privacy. I'll leave it up to you to evaluate what is and is not appropriate here for this new and incredibly powerful new analysis tool.

For More Help

Additional **PostScript**, **Acrobat**, **eBay**, and **Webmastering** assistance is available per the previously shown web links. Custom modification and design services are available at our standard consulting rates. Per our **InfoPack Services**. Or you can directly **email** me.

Additional **GuruGrams** columns await your ongoing support as a **Synergetics Partner**.