

🔑 84f203fb1c ▾

⋮

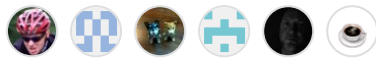
[tinyproxy](#) / [src](#) / [reqs.c](#)



rofl0r fix reversepath directive using https url giving misleading error ... ✓

🕒 History

👤 6 contributors



1787 lines (1544 sloc) | 61.8 KB

⋮

```

1  /* tinyproxy - A fast light-weight HTTP proxy
2  * Copyright (C) 1998 Steven Young <sdyoung@miranda.org>
3  * Copyright (C) 1999-2005 Robert James Kaes <rjkaes@users.sourceforge.net>
4  * Copyright (C) 2000 Chris Lightfoot <chris@ex-parrot.com>
5  * Copyright (C) 2002 Petr Lampa <lampa@fit.vutbr.cz>
6  *
7  * This program is free software; you can redistribute it and/or modify
8  * it under the terms of the GNU General Public License as published by
9  * the Free Software Foundation; either version 2 of the License, or
10 * (at your option) any later version.
11 *
12 * This program is distributed in the hope that it will be useful,
13 * but WITHOUT ANY WARRANTY; without even the implied warranty of
14 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15 * GNU General Public License for more details.
16 *
17 * You should have received a copy of the GNU General Public License along
18 * with this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22 /* This is where all the work in tinyproxy is actually done. Incoming
23 * connections have a new child created for them. The child then
24 * processes the headers from the client, the response from the server,
25 * and then relays the bytes between the two.
26 */
27
28 #include "main.h"
29

```

```

30 #include "acl.h"
31 #include "anonymous.h"
32 #include "buffer.h"
33 #include "conns.h"
34 #include "filter.h"
35 #include "hsearch.h"
36 #include "orderedmap.h"
37 #include "heap.h"
38 #include "html-error.h"
39 #include "log.h"
40 #include "network.h"
41 #include "reqs.h"
42 #include "sock.h"
43 #include "stats.h"
44 #include "text.h"
45 #include "utils.h"
46 #include "sblst.h"
47 #include "reverse-proxy.h"
48 #include "transparent-proxy.h"
49 #include "upstream.h"
50 #include "connect-ports.h"
51 #include "conf.h"
52 #include "basicauth.h"
53 #include "loop.h"
54 #include "mypoll.h"
55
56 /*
57  * Maximum length of a HTTP line
58  */
59 #define HTTP_LINE_LENGTH (MAXBUFSIZE / 6)
60
61 /*
62  * Macro to help test if the Upstream proxy supported is compiled in and
63  * enabled.
64  */
65 #ifdef UPSTREAM_SUPPORT
66 # define UPSTREAM_CONFIGURED() (config->upstream_list != NULL)
67 # define UPSTREAM_HOST(host) upstream_get(host, config->upstream_list)
68 # define UPSTREAM_IS_HTTP(conn) (conn->upstream_proxy != NULL && conn->upstream_proxy->type == PT
69 #else
70 # define UPSTREAM_CONFIGURED() (0)
71 # define UPSTREAM_HOST(host) (NULL)
72 # define UPSTREAM_IS_HTTP(up) (0)
73 #endif
74
75 /*
76  * Codify the test for the carriage return and new line characters.
77  */
78 #define CHECK_CRLF(header, len) \

```

```

79     ((len) == 1 && header[0] == '\n') || \
80     ((len) == 2 && header[0] == '\r' && header[1] == '\n'))
81
82 /*
83  * Codify the test for header fields folded over multiple lines.
84  */
85 #define CHECK_LWS(header, len) \
86     ((len) > 0 && (header[0] == ' ' || header[0] == '\t'))
87
88 /*
89  * Read in the first line from the client (the request line for HTTP
90  * connections. The request line is allocated from the heap, but it must
91  * be freed in another function.
92  */
93 static int read_request_line (struct conn_s *connptr)
94 {
95     ssize_t len;
96
97     retry:
98     len = readline (connptr->client_fd, &connptr->request_line);
99     if (len <= 0) {
100         log_message (LOG_ERR,
101                     "read_request_line: Client (file descriptor: %d) "
102                     "closed socket before read.", connptr->client_fd);
103
104         return -1;
105     }
106
107     /*
108      * Strip the new line and carriage return from the string.
109      */
110     if (chomp (connptr->request_line, len) == len) {
111         /*
112          * If the number of characters removed is the same as the
113          * length then it was a blank line. Free the buffer and
114          * try again (since we're looking for a request line.)
115          */
116         safefree (connptr->request_line);
117         goto retry;
118     }
119
120     log_message (LOG_CONN, "Request (file descriptor %d): %s",
121                 connptr->client_fd, connptr->request_line);
122
123     return 0;
124 }
125
126 /*
127  * Free all the memory allocated in a request.

```

```

128  */
129  static void free_request_struct (struct request_s *request)
130  {
131      if (!request)
132          return;
133
134      safefree (request->method);
135      safefree (request->protocol);
136
137      if (request->host)
138          safefree (request->host);
139      if (request->path)
140          safefree (request->path);
141
142      safefree (request);
143  }
144
145  /*
146   * Take a host string and if there is a username/password part, strip
147   * it off.
148   */
149  static void strip_username_password (char *host)
150  {
151      char *p;
152
153      assert (host);
154      assert (strlen (host) > 0);
155
156      if ((p = strchr (host, '@')) == NULL)
157          return;
158
159      /*
160       * Move the pointer past the "@" and then copy from that point
161       * until the NUL to the beginning of the host buffer.
162       */
163      p++;
164      while (*p)
165          *host++ = *p++;
166      *host = '\0';
167  }
168
169  /*
170   * Take a host string and if there is a port part, strip
171   * it off and set proper port variable i.e. for www.host.com:8001
172   */
173  static int strip_return_port (char *host)
174  {
175      char *ptr1;
176      char *ptr2;

```

```

177     int port;
178
179     ptr1 = strrchr (host, ':');
180     if (ptr1 == NULL)
181         return 0;
182
183     /* Check for IPv6 style literals */
184     ptr2 = strchr (ptr1, '[');
185     if (ptr2 != NULL)
186         return 0;
187
188     *ptr1++ = '\\0';
189     if (sscanf (ptr1, "%d", &port) != 1)    /* one conversion required */
190         return 0;
191     return port;
192 }
193
194 /*
195  * Pull the information out of the URL line.
196  * This expects urls with the initial '<proto>://'
197  * part stripped and hence can handle http urls,
198  * (proxied) ftp:// urls and https-requests that
199  * come in without the proto:// part via CONNECT.
200  */
201 static int extract_url (const char *url, int default_port,
202                        struct request_s *request)
203 {
204     char *p;
205     int port;
206
207     /* Split the URL on the slash to separate host from path */
208     p = strchr (url, '/');
209     if (p != NULL) {
210         int len;
211         len = p - url;
212         request->host = (char *) safemalloc (len + 1);
213         memcpy (request->host, url, len);
214         request->host[len] = '\\0';
215         request->path = safestrdup (p);
216     } else {
217         request->host = safestrdup (url);
218         request->path = safestrdup ("/");
219     }
220
221     if (!request->host || !request->path)
222         goto ERROR_EXIT;
223
224     /* Remove the username/password if they're present */
225     strip_username_password (request->host);

```

```

226
227     /* Find a proper port in www.site.com:8001 URLs */
228     port = strip_return_port (request->host);
229     request->port = (port != 0) ? port : default_port;
230
231     /* Remove any surrounding '[' and ']' from IPv6 literals */
232     p = strrchr (request->host, '[');
233     if (p && (*(request->host) == '[')) {
234         memmove(request->host, request->host + 1,
235                 strlen(request->host) - 2);
236         *p = '\\0';
237         p--;
238         *p = '\\0';
239     }
240
241     return 0;
242
243 ERROR_EXIT:
244     if (request->host)
245         safefree (request->host);
246     if (request->path)
247         safefree (request->path);
248
249     return -1;
250 }
251
252 /*
253  * Create a connection for HTTP connections.
254  */
255 static int
256 establish_http_connection (struct conn_s *connptr, struct request_s *request)
257 {
258     char portbuff[7];
259     char dst[sizeof(struct in6_addr)];
260
261     /* Build a port string if it's not a standard port */
262     if (request->port != HTTP_PORT && request->port != HTTP_PORT_SSL)
263         snprintf (portbuff, 7, "%u", request->port);
264     else
265         portbuff[0] = '\\0';
266
267     if (inet_pton(AF_INET6, request->host, dst) > 0) {
268         /* host is an IPv6 address literal, so surround it with
269          * [] */
270         return write_message (connptr->server_fd,
271                               "%s %s HTTP/1.%u\\r\\n"
272                               "Host: [%s]%s\\r\\n"
273                               "Connection: close\\r\\n",
274                               request->method, request->path,

```

```

275         connptr->protocol.major != 1 ? 0 :
276             connptr->protocol.minor,
277         request->host, portbuff);
278     } else if (connptr->upstream_proxy &&
279         connptr->upstream_proxy->type == PT_HTTP &&
280         connptr->upstream_proxy->ua.authstr) {
281         return write_message (connptr->server_fd,
282             "%s %s HTTP/1.%u\r\n"
283             "Host: %s%s\r\n"
284             "Connection: close\r\n"
285             "Proxy-Authorization: Basic %s\r\n",
286             request->method, request->path,
287             connptr->protocol.major != 1 ? 0 :
288                 connptr->protocol.minor,
289             request->host, portbuff,
290             connptr->upstream_proxy->ua.authstr);
291     } else {
292         return write_message (connptr->server_fd,
293             "%s %s HTTP/1.%u\r\n"
294             "Host: %s%s\r\n"
295             "Connection: close\r\n",
296             request->method, request->path,
297             connptr->protocol.major != 1 ? 0 :
298                 connptr->protocol.minor,
299             request->host, portbuff);
300     }
301 }
302
303 /*
304  * Send the appropriate response to the client to establish a
305  * connection via CONNECT method.
306  */
307 static int send_connect_method_response (struct conn_s *connptr)
308 {
309     return write_message (connptr->client_fd,
310         "HTTP/1.%u 200 Connection established\r\n"
311         "Proxy-agent: " PACKAGE "/" VERSION "\r\n"
312         "\r\n", connptr->protocol.major != 1 ? 0 :
313             connptr->protocol.minor);
314 }
315
316 /*
317  * Break the request line apart and figure out where to connect and
318  * build a new request line. Finally connect to the remote server.
319  */
320 static struct request_s *process_request (struct conn_s *connptr,
321     orderedmap hashofheaders)
322 {
323     char *url;

```

```

324     struct request_s *request;
325     int ret, skip_trans;
326     size_t request_len;
327
328     skip_trans = 0;
329
330     /* NULL out all the fields so frees don't cause segfaults. */
331     request =
332         (struct request_s *) safecalloc (1, sizeof (struct request_s));
333     if (!request)
334         return NULL;
335
336     request_len = strlen (connptr->request_line) + 1;
337
338     request->method = (char *) safemalloc (request_len);
339     url = (char *) safemalloc (request_len);
340     request->protocol = (char *) safemalloc (request_len);
341
342     if (!request->method || !url || !request->protocol) {
343         goto fail;
344     }
345
346     ret = sscanf (connptr->request_line, "%[^ ] %[^ ] %[^ ]",
347                  request->method, url, request->protocol);
348     if (ret == 2 && !strcasecmp (request->method, "GET")) {
349         request->protocol[0] = 0;
350
351         /* Indicate that this is a HTTP/0.9 GET request */
352         connptr->protocol.major = 0;
353         connptr->protocol.minor = 9;
354     } else if (ret == 3 && !strncasecmp (request->protocol, "HTTP/", 5)) {
355         /*
356          * Break apart the protocol and update the connection
357          * structure.
358          */
359         ret = sscanf (request->protocol + 5, "%u.%u",
360                      &connptr->protocol.major,
361                      &connptr->protocol.minor);
362
363         /*
364          * If the conversion doesn't succeed, drop down below and
365          * send the error to the user.
366          */
367         if (ret != 2)
368             goto BAD_REQUEST_ERROR;
369     } else {
370 BAD_REQUEST_ERROR:
371         log_message (LOG_ERR,
372                     "process_request: Bad Request on file descriptor %d",

```



```

373         connptr->client_fd);
374     indicate_http_error (connptr, 400, "Bad Request",
375         "detail", "Request has an invalid format",
376         "url", url, NULL);
377     goto fail;
378 }
379
380 #ifdef REVERSE_SUPPORT
381     if (config->reversepath_list != NULL) {
382         /*
383          * Rewrite the URL based on the reverse path. After calling
384          * reverse_rewrite_url "url" can be freed since we either
385          * have the newly rewritten URL, or something failed and
386          * we'll be closing anyway.
387          */
388         char *reverse_url;
389         int reverse_status;
390
391         reverse_url = reverse_rewrite_url (connptr, hashofheaders, url, &reverse_status);
392
393         if (reverse_url != NULL) {
394             if (reverse_status == 301) {
395                 char buf[PATH_MAX];
396                 snprintf (buf, sizeof buf, "Location: %s\r\n", reverse_url);
397                 send_http_headers (connptr, 301, "Moved Permanently", buf);
398                 goto fail;
399             }
400             safefree (url);
401             url = reverse_url;
402             skip_trans = 1;
403         } else if (config->reverseonly) {
404             log_message (LOG_ERR,
405                 "Bad request, no mapping for '%s' found",
406                 url);
407             indicate_http_error (connptr, 400, "Bad Request",
408                 "detail", "No mapping found for "
409                 "requested url", "url", url, NULL);
410             goto fail;
411         }
412     }
413 #endif
414
415     if (strncasecmp (url, "http://", 7) == 0
416         || (UPSTREAM_CONFIGURED () && strncasecmp (url, "ftp://", 6) == 0))
417     {
418         char *skipped_type = strstr (url, "///") + 2;
419
420         if (extract_url (skipped_type, HTTP_PORT, request) < 0) {
421             indicate_http_error (connptr, 400, "Bad Request",

```

```

422         "detail", "Could not parse URL",
423         "url", url, NULL);
424         goto fail;
425     }
426     } else if (strcmp (request->method, "CONNECT") == 0) {
427         if (extract_url (url, HTTP_PORT_SSL, request) < 0) {
428             indicate_http_error (connptr, 400, "Bad Request",
429             "detail", "Could not parse URL",
430             "url", url, NULL);
431             goto fail;
432         }
433
434         /* Verify that the port in the CONNECT method is allowed */
435         if (!check_allowed_connect_ports (request->port,
436             config->connect_ports))
437         {
438             indicate_http_error (connptr, 403, "Access violation",
439             "detail",
440             "The CONNECT method not allowed "
441             "with the port you tried to use.",
442             "url", url, NULL);
443             log_message (LOG_INFO,
444             "Refused CONNECT method on port %d",
445             request->port);
446             goto fail;
447         }
448
449         connptr->connect_method = TRUE;
450     } else {
451 #ifdef TRANSPARENT_PROXY
452         if (!skip_trans) {
453             if (!do_transparent_proxy
454                 (connptr, hashofheaders, request, config, &url))
455                 goto fail;
456         } else
457 #endif
458         {
459             indicate_http_error (connptr, 501, "Not Implemented",
460             "detail",
461             "Unknown method or unsupported protocol.",
462             "url", url, NULL);
463             log_message (LOG_INFO, "Unknown method (%s) or protocol (%s)",
464             request->method, url);
465             goto fail;
466         }
467     }
468
469 #ifdef FILTER_ENABLE
470     /*

```

```

471     * Filter restricted domains/urls
472     */
473     if (config->filter) {
474         int fu = config->filter_opts & FILTER_OPT_URL;
475         ret = filter_run (fu ? url : request->host);
476
477         if (ret) {
478             update_stats (STAT_DENIED);
479
480             log_message (LOG_NOTICE,
481                 "Proxying refused on filtered %s \"%s\"",
482                 fu ? "url" : "domain",
483                 fu ? url : request->host);
484
485             indicate_http_error (connptr, 403, "Filtered",
486                 "detail",
487                 "The request you made has been filtered",
488                 "url", url, NULL);
489             goto fail;
490         }
491     }
492 #endif
493
494
495     /*
496     * Check to see if they're requesting the stat host
497     */
498     if (config->stathost && strcmp (config->stathost, request->host) == 0) {
499         log_message (LOG_NOTICE, "Request for the stathost.");
500         connptr->show_stats = TRUE;
501         goto fail;
502     }
503
504     safefree (url);
505
506     return request;
507
508 fail:
509     safefree (url);
510     free_request_struct (request);
511     return NULL;
512 }
513
514 /*
515  * pull_client_data is used to pull across any client data (like in a
516  * POST) which needs to be handled before an error can be reported, or
517  * server headers can be processed.
518  *     - rjkaes
519  */

```

```

520 static int pull_client_data (struct conn_s *connptr, long int length, int iehack)
521 {
522     char *buffer;
523     ssize_t len;
524     int ret;
525
526     buffer =
527         (char *) safemalloc (min (MAXBUFSIZE, (unsigned long int) length));
528     if (!buffer)
529         return -1;
530
531     do {
532         len = safe_read (connptr->client_fd, buffer,
533                         min (MAXBUFSIZE, (unsigned long int) length));
534         if (len <= 0)
535             goto ERROR_EXIT;
536
537         if (!connptr->error_variables) {
538             if (safe_write (connptr->server_fd, buffer, len) < 0)
539                 goto ERROR_EXIT;
540         }
541
542         length -= len;
543     } while (length > 0);
544
545     if (iehack) {
546         /*
547          * BUG FIX: Internet Explorer will leave two bytes (carriage
548          * return and line feed) at the end of a POST message. These
549          * need to be eaten for tinypoxy to work correctly.
550          */
551         ret = socket_nonblocking (connptr->client_fd);
552         if (ret != 0) {
553             log_message(LOG_ERR, "Failed to set the client socket "
554                         "to non-blocking: %s", strerror(errno));
555             goto ERROR_EXIT;
556         }
557
558         len = recv (connptr->client_fd, buffer, 2, MSG_PEEK);
559
560         ret = socket_blocking (connptr->client_fd);
561         if (ret != 0) {
562             log_message(LOG_ERR, "Failed to set the client socket "
563                         "to blocking: %s", strerror(errno));
564             goto ERROR_EXIT;
565         }
566
567         if (len < 0 && errno != EAGAIN)
568             goto ERROR_EXIT;

```

```

569
570         if ((len == 2) && CHECK_CRLF (buffer, len)) {
571             ssize_t bytes_read;
572
573             bytes_read = read (connptr->client_fd, buffer, 2);
574             if (bytes_read == -1) {
575                 log_message
576                     (LOG_WARNING,
577                      "Could not read two bytes from POST message");
578             }
579         }
580     }
581
582     safefree (buffer);
583     return 0;
584
585 ERROR_EXIT:
586     safefree (buffer);
587     return -1;
588 }
589
590 /* pull chunked client data */
591 static int pull_client_data_chunked (struct conn_s *connptr) {
592     char *buffer = 0;
593     ssize_t len;
594     long chunklen;
595
596     while(1) {
597         if (buffer) safefree(buffer);
598         len = readline (connptr->client_fd, &buffer);
599
600         if (len <= 0)
601             goto ERROR_EXIT;
602
603         if (!connptr->error_variables) {
604             if (safe_write (connptr->server_fd, buffer, len) < 0)
605                 goto ERROR_EXIT;
606         }
607
608         chunklen = strtol (buffer, (char**)0, 16);
609
610         if (pull_client_data (connptr, chunklen+2, 0) < 0)
611             goto ERROR_EXIT;
612
613         if(!chunklen) break;
614     }
615
616     safefree (buffer);
617     return 0;

```

```

618
619 ERROR_EXIT:
620     safefree (buffer);
621     return -1;
622 }
623
624 #ifdef XTINYPROXY_ENABLE
625 /*
626  * Add the X-Tinyproxy header to the collection of headers being sent to
627  * the server.
628  *     -rjkaes
629  */
630 static int add_xtinyproxy_header (struct conn_s *connptr)
631 {
632     assert (connptr && connptr->server_fd >= 0);
633     return write_message (connptr->server_fd,
634                          "X-Tinyproxy: %s\r\n", connptr->client_ip_addr);
635 }
636 #endif /* XTINYPROXY */
637
638 /*
639  * Take a complete header line and break it apart (into a key and the data.)
640  * Now insert this information into the hashmap for the connection so it
641  * can be retrieved and manipulated later.
642  */
643 static int
644 add_header_to_connection (orderedmap hashofheaders, char *header, size_t len)
645 {
646     char *sep;
647
648     /* Get rid of the new line and return at the end */
649     len -= chomp (header, len);
650
651     sep = strchr (header, ':');
652     if (!sep)
653         return 0; /* just skip invalid header, do not give error */
654
655     /* Blank out colons, spaces, and tabs. */
656     while (*sep == ':' || *sep == ' ' || *sep == '\t')
657         *sep++ = '\0';
658
659     /* Calculate the new length of just the data */
660     len -= sep - header - 1;
661
662     return orderedmap_append (hashofheaders, header, sep);
663 }
664
665 /*
666  * Define maximum number of headers that we accept.

```

```

667  * This should be big enough to handle legitimate cases,
668  * but limited to avoid DoS.
669  */
670  #define MAX_HEADERS 10000
671
672  /*
673   * Read all the headers from the stream
674   */
675  static int get_all_headers (int fd, orderedmap hashofheaders)
676  {
677      char *line = NULL;
678      char *header = NULL;
679      int count;
680      char *tmp;
681      ssize_t linelen;
682      ssize_t len = 0;
683      unsigned int double_cgi = FALSE;      /* boolean */
684
685      assert (fd >= 0);
686      assert (hashofheaders != NULL);
687
688      for (count = 0; count < MAX_HEADERS; count++) {
689          if ((linelen = readline (fd, &line)) <= 0) {
690              safefree (header);
691              safefree (line);
692              return -1;
693          }
694
695          /*
696           * If we received a CR LF or a non-continuation line, then add
697           * the accumulated header field, if any, to the hashmap, and
698           * reset it.
699           */
700          if (CHECK_CRLF (line, linelen) || !CHECK_LWS (line, linelen)) {
701              if (!double_cgi
702                  && len > 0
703                  && add_header_to_connection (hashofheaders, header,
704                                                len) < 0) {
705                  safefree (header);
706                  safefree (line);
707                  return -1;
708              }
709
710              len = 0;
711          }
712
713          /*
714           * If we received just a CR LF on a line, the headers are
715           * finished.

```

```

716         */
717         if (CHECK_CRLF (line, linelen)) {
718             safefree (header);
719             safefree (line);
720             return 0;
721         }
722
723         /*
724          * BUG FIX: The following code detects a "Double CGI"
725          * situation so that we can handle the nonconforming system.
726          * This problem was found when accessing cgi.ebay.com, and it
727          * turns out to be a wider spread problem as well.
728          *
729          * If "Double CGI" is in effect, duplicate headers are
730          * ignored.
731          *
732          * FIXME: Might need to change this to a more robust check.
733          */
734         if (linelen >= 5 && strncasecmp (line, "HTTP/", 5) == 0) {
735             double_cgi = TRUE;
736         }
737
738         /*
739          * Append the new line to the current header field.
740          */
741         tmp = (char *) saferealloc (header, len + linelen);
742         if (tmp == NULL) {
743             safefree (header);
744             safefree (line);
745             return -1;
746         }
747
748         header = tmp;
749         memcpy (header + len, line, linelen);
750         len += linelen;
751
752         safefree (line);
753     }
754
755     /*
756      * If we get here, this means we reached MAX_HEADERS count.
757      * Bail out with error.
758      */
759     safefree (header);
760     safefree (line);
761     return -1;
762 }
763
764 /*

```



```

765  * Extract the headers to remove. These headers were listed in the Connection
766  * and Proxy-Connection headers.
767  */
768  static int remove_connection_headers (orderedmap hashofheaders)
769  {
770      static const char *headers[] = {
771          "connection",
772          "proxy-connection"
773      };
774
775      char *data;
776      char *ptr;
777      ssize_t len;
778      int i;
779
780      for (i = 0; i != (sizeof (headers) / sizeof (char *)); ++i) {
781          /* Look for the connection header. If it's not found, return. */
782          data = orderedmap_find(hashofheaders, headers[i]);
783
784          if (!data)
785              return 0;
786
787          len = strlen(data);
788
789          /*
790           * Go through the data line and replace any special characters
791           * with a NULL.
792           */
793          ptr = data;
794          while ((ptr = strpbrk (ptr, "()<>@,;:\\""/[]?={ } \t"))
795                  *ptr++ = '\0');
796
797          /*
798           * All the tokens are separated by NULLs. Now go through the
799           * token and remove them from the hashofheaders.
800           */
801          ptr = data;
802          while (ptr < data + len) {
803              orderedmap_remove (hashofheaders, ptr);
804
805              /* Advance ptr to the next token */
806              ptr += strlen (ptr) + 1;
807              while (ptr < data + len && *ptr == '\0')
808                  ptr++;
809          }
810
811          /* Now remove the connection header it self. */
812          orderedmap_remove (hashofheaders, headers[i]);
813      }

```

```

814
815     return 0;
816 }
817
818 /*
819  * If there is a Content-Length header, then return the value; otherwise, return
820  * -1.
821  */
822 static long get_content_length (orderedmap hashofheaders)
823 {
824     char *data;
825     long content_length = -1;
826
827     data = orderedmap_find (hashofheaders, "content-length");
828
829     if (data)
830         content_length = atol (data);
831
832     return content_length;
833 }
834
835 static int is_chunked_transfer (orderedmap hashofheaders)
836 {
837     char *data;
838     data = orderedmap_find (hashofheaders, "transfer-encoding");
839     return data ? !strcmp (data, "chunked") : 0;
840 }
841
842 /*
843  * Search for Via header in a hash of headers and either write a new Via
844  * header, or append our information to the end of an existing Via header.
845  *
846  * FIXME: Need to add code to "hide" our internal information for security
847  * purposes.
848  */
849 static int
850 write_via_header (int fd, orderedmap hashofheaders,
851                  unsigned int major, unsigned int minor)
852 {
853     char hostname[512];
854     char *data;
855     int ret;
856
857     if (config->disable_viaheader) {
858         ret = 0;
859         goto done;
860     }
861
862     if (config->via_proxy_name) {

```

```

863         strcpy (hostname, config->via_proxy_name, sizeof (hostname));
864     } else if (gethostname (hostname, sizeof (hostname)) < 0) {
865         strcpy (hostname, "unknown", 512);
866     }
867
868     /*
869     * See if there is a "Via" header. If so, again we need to do a bit
870     * of processing.
871     */
872     data = orderedmap_find (hashofheaders, "via");
873     if (data) {
874         ret = write_message (fd,
875                             "Via: %s, %hu.%hu %s (%s/%s)\r\n",
876                             data, major, minor, hostname, PACKAGE,
877                             VERSION);
878
879         orderedmap_remove (hashofheaders, "via");
880     } else {
881         ret = write_message (fd,
882                             "Via: %hu.%hu %s (%s/%s)\r\n",
883                             major, minor, hostname, PACKAGE, VERSION);
884     }
885
886 done:
887     return ret;
888 }
889
890 /*
891 * Number of buckets to use internally in the hashmap.
892 */
893 #define HEADER_BUCKETS 32
894
895 /*
896 * Here we loop through all the headers the client is sending. If we
897 * are running in anonymous mode, we will _only_ send the headers listed
898 * (plus a few which are required for various methods).
899 *     - rjkaes
900 */
901 static int
902 process_client_headers (struct conn_s *connptr, orderedmap hashofheaders)
903 {
904     static const char *skipheaders[] = {
905         "host",
906         "keep-alive",
907         "proxy-connection",
908         "te",
909         "trailers",
910         "upgrade"
911     };

```

```

912     int i;
913     size_t iter;
914     int ret = 0;
915
916     char *data, *header;
917
918     /*
919      * Don't send headers if there's already an error, if the request was
920      * a stats request, or if this was a CONNECT method (unless upstream
921      * http proxy is in use.)
922      */
923     if (connptr->server_fd == -1 || connptr->show_stats
924         || (connptr->connect_method && ! UPSTREAM_IS_HTTP(connptr))) {
925         log_message (LOG_INFO,
926                     "Not sending client headers to remote machine");
927         return 0;
928     }
929
930     /*
931      * See if there is a "Content-Length" header.  If so, again we need
932      * to do a bit of processing.
933      */
934     connptr->content_length.client = get_content_length (hashofheaders);
935
936     /* Check whether client sends chunked data. */
937     if (connptr->content_length.client == -1 && is_chunked_transfer (hashofheaders))
938         connptr->content_length.client = -2;
939
940     /*
941      * See if there is a "Connection" header.  If so, we need to do a bit
942      * of processing. :)
943      */
944     remove_connection_headers (hashofheaders);
945
946     /*
947      * Delete the headers listed in the skipheaders list
948      */
949     for (i = 0; i != (sizeof (skipheaders) / sizeof (char *)); i++) {
950         orderedmap_remove (hashofheaders, skipheaders[i]);
951     }
952
953     /* Send, or add the Via header */
954     ret = write_via_header (connptr->server_fd, hashofheaders,
955                            connptr->protocol.major,
956                            connptr->protocol.minor);
957     if (ret < 0) {
958         indicate_http_error (connptr, 503,
959                             "Could not send data to remote server",
960                             "detail",

```

```

961         "A network error occurred while "
962         "trying to write data to the remote web server.",
963         NULL);
964     goto PULL_CLIENT_DATA;
965 }
966
967 /*
968  * Output all the remaining headers to the remote machine.
969  */
970 iter = 0;
971 while((iter = orderedmap_next(hashofheaders, iter, &data, &header))) {
972     if (!is_anonymous_enabled (config)
973         || anonymous_search (config, data) > 0) {
974         ret =
975             write_message (connptr->server_fd,
976                           "%s: %s\r\n", data, header);
977         if (ret < 0) {
978             indicate_http_error (connptr, 503,
979                                  "Could not send data to remote server",
980                                  "detail",
981                                  "A network error occurred while "
982                                  "trying to write data to the "
983                                  "remote web server.",
984                                  NULL);
985             goto PULL_CLIENT_DATA;
986         }
987     }
988 }
989 #if defined(XTINYPROXY_ENABLE)
990     if (config->add_xtinyproxy)
991         add_xtinyproxy_header (connptr);
992 #endif
993
994 /* Write the final "blank" line to signify the end of the headers */
995 if (safe_write (connptr->server_fd, "\r\n", 2) < 0)
996     return -1;
997
998 /*
999  * Spin here pulling the data from the client.
1000  */
1001 PULL_CLIENT_DATA:
1002     if (connptr->content_length.client > 0) {
1003         ret = pull_client_data (connptr,
1004                                connptr->content_length.client, 1);
1005     } else if (connptr->content_length.client == -2)
1006         ret = pull_client_data_chunked (connptr);
1007
1008     return ret;
1009 }

```

```

1010
1011 /*
1012  * Loop through all the headers (including the response code) from the
1013  * server.
1014  */
1015 static int process_server_headers (struct conn_s *connptr)
1016 {
1017     static const char *skipheaders[] = {
1018         "keep-alive",
1019         "proxy-authenticate",
1020         "proxy-authorization",
1021         "proxy-connection",
1022     };
1023
1024     char *response_line;
1025
1026     orderedmap hashofheaders;
1027     size_t iter;
1028     char *data, *header;
1029     ssize_t len;
1030     int i;
1031     int ret;
1032
1033 #ifdef REVERSE_SUPPORT
1034     struct reversepath *reverse = config->reversepath_list;
1035 #endif
1036
1037     /* Get the response line from the remote server. */
1038 retry:
1039     len = readline (connptr->server_fd, &response_line);
1040     if (len <= 0)
1041         return -1;
1042
1043     /*
1044      * Strip the new line and character return from the string.
1045      */
1046     if (chomp (response_line, len) == len) {
1047         /*
1048          * If the number of characters removed is the same as the
1049          * length then it was a blank line. Free the buffer and
1050          * try again (since we're looking for a request line.)
1051          */
1052         safefree (response_line);
1053         goto retry;
1054     }
1055
1056     hashofheaders = orderedmap_create (HEADER_BUCKETS);
1057     if (!hashofheaders) {
1058         safefree (response_line);

```

```

1059         return -1;
1060     }
1061
1062     /*
1063     * Get all the headers from the remote server in a big hash
1064     */
1065     if (get_all_headers (connptr->server_fd, hashofheaders) < 0) {
1066         log_message (LOG_WARNING,
1067             "Could not retrieve all the headers from the remote server.");
1068         orderedmap_destroy (hashofheaders);
1069         safefree (response_line);
1070
1071         indicate_http_error (connptr, 503,
1072             "Could not retrieve all the headers",
1073             "detail",
1074             PACKAGE_NAME " "
1075             "was unable to retrieve and process headers from "
1076             "the remote web server.", NULL);
1077         return -1;
1078     }
1079
1080     /*
1081     * At this point we've received the response line and all the
1082     * headers. However, if this is a simple HTTP/0.9 request we
1083     * CAN NOT send any of that information back to the client.
1084     * Instead we'll free all the memory and return.
1085     */
1086     if (connptr->protocol.major < 1) {
1087         orderedmap_destroy (hashofheaders);
1088         safefree (response_line);
1089         return 0;
1090     }
1091
1092     /* Send the saved response line first */
1093     ret = write_message (connptr->client_fd, "%s\r\n", response_line);
1094     safefree (response_line);
1095     if (ret < 0)
1096         goto ERROR_EXIT;
1097
1098     /*
1099     * If there is a "Content-Length" header, retrieve the information
1100     * from it for later use.
1101     */
1102     connptr->content_length.server = get_content_length (hashofheaders);
1103
1104     /*
1105     * See if there is a connection header. If so, we need to to a bit of
1106     * processing.
1107     */

```

```

1108     remove_connection_headers (hashofheaders);
1109
1110     /*
1111      * Delete the headers listed in the skipheaders list
1112      */
1113     for (i = 0; i != (sizeof (skipheaders) / sizeof (char *)); i++) {
1114         orderedmap_remove (hashofheaders, skipheaders[i]);
1115     }
1116
1117     /* Send, or add the Via header */
1118     ret = write_via_header (connptr->client_fd, hashofheaders,
1119                             connptr->protocol.major,
1120                             connptr->protocol.minor);
1121
1122     if (ret < 0)
1123         goto ERROR_EXIT;
1124
1125 #ifndef REVERSE_SUPPORT
1126     /* Write tracking cookie for the magical reverse proxy path hack */
1127     if (config->reversemagic && connptr->reversepath) {
1128         ret = write_message (connptr->client_fd,
1129                             "Set-Cookie: " REVERSE_COOKIE
1130                             "=%s; path=/%r\n", connptr->reversepath);
1131
1132         if (ret < 0)
1133             goto ERROR_EXIT;
1134     }
1135
1136     /* Rewrite the HTTP redirect if needed */
1137     if (config->reversebaseurl &&
1138         (header = orderedmap_find (hashofheaders, "location"))) {
1139
1140         /* Look for a matching entry in the reversepath list */
1141         while (reverse) {
1142             if (strncasecmp (header,
1143                             reverse->url, (len =
1144                                 strlen (reverse->
1145                                     url))) == 0)
1146                 break;
1147             reverse = reverse->next;
1148         }
1149
1150         if (reverse) {
1151             ret =
1152                 write_message (connptr->client_fd,
1153                             "Location: %s%s%s\r\n",
1154                             config->reversebaseurl,
1155                             (reverse->path + 1), (header + len));
1156
1157             if (ret < 0)
1158                 goto ERROR_EXIT;
1159         }
1160     }

```



```

1157         log_message (LOG_INFO,
1158             "Rewriting HTTP redirect: %s -> %s%s%s",
1159             header, config->reversebaseurl,
1160             (reverse->path + 1), (header + len));
1161         orderedmap_remove (hashofheaders, "location");
1162     }
1163 }
1164 #endif
1165
1166 /*
1167  * All right, output all the remaining headers to the client.
1168  */
1169 iter = 0;
1170 while ((iter = orderedmap_next(hashofheaders, iter, &data, &header))) {
1171
1172     ret = write_message (connptr->client_fd,
1173         "%s: %s\r\n", data, header);
1174     if (ret < 0)
1175         goto ERROR_EXIT;
1176 }
1177 orderedmap_destroy (hashofheaders);
1178
1179 /* Write the final blank line to signify the end of the headers */
1180 if (safe_write (connptr->client_fd, "\r\n", 2) < 0)
1181     return -1;
1182
1183 return 0;
1184
1185 ERROR_EXIT:
1186     orderedmap_destroy (hashofheaders);
1187     return -1;
1188 }
1189
1190 /*
1191  * Switch the sockets into nonblocking mode and begin relaying the bytes
1192  * between the two connections. We continue to use the buffering code
1193  * since we want to be able to buffer a certain amount for slower
1194  * connections (as this was the reason why I originally modified
1195  * tinyproxy oh so long ago...)
1196  * - rjkaes
1197  */
1198 static void relay_connection (struct conn_s *connptr)
1199 {
1200     int ret;
1201     ssize_t bytes_received;
1202
1203     for (;;) {
1204         pollfd_struct fds[2] = {0};
1205         fds[0].fd = connptr->client_fd;

```

```

1206         fds[1].fd = connptr->server_fd;
1207
1208         if (buffer_size (connptr->sbuffer) > 0)
1209             fds[0].events |= MYPoll_WRITE;
1210         if (buffer_size (connptr->cbuffer) > 0)
1211             fds[1].events |= MYPoll_WRITE;
1212         if (buffer_size (connptr->sbuffer) < MAXBUFSIZE)
1213             fds[1].events |= MYPoll_READ;
1214         if (buffer_size (connptr->cbuffer) < MAXBUFSIZE)
1215             fds[0].events |= MYPoll_READ;
1216
1217         ret = mypoll(fds, 2, config->idletimeout);
1218
1219         if (ret == 0) {
1220             log_message (LOG_INFO,
1221                 "Idle Timeout (after " SELECT_OR_POLL ")");
1222             return;
1223         } else if (ret < 0) {
1224             log_message (LOG_ERR,
1225                 "relay_connection: " SELECT_OR_POLL "() error \"%s\". "
1226                 "Closing connection (client_fd:%d, server_fd:%d)",
1227                 strerror (errno), connptr->client_fd,
1228                 connptr->server_fd);
1229             return;
1230         }
1231
1232         if (fds[1].revents & MYPoll_READ) {
1233             bytes_received =
1234                 read_buffer (connptr->server_fd, connptr->sbuffer);
1235             if (bytes_received < 0)
1236                 break;
1237
1238             connptr->content_length.server -= bytes_received;
1239             if (connptr->content_length.server == 0)
1240                 break;
1241         }
1242         if ((fds[0].revents & MYPoll_READ)
1243             && read_buffer (connptr->client_fd, connptr->cbuffer) < 0) {
1244             break;
1245         }
1246         if ((fds[1].revents & MYPoll_WRITE)
1247             && write_buffer (connptr->server_fd, connptr->cbuffer) < 0) {
1248             break;
1249         }
1250         if ((fds[0].revents & MYPoll_WRITE)
1251             && write_buffer (connptr->client_fd, connptr->sbuffer) < 0) {
1252             break;
1253         }
1254     }

```

```

1255
1256     while (buffer_size (connptr->sbuffer) > 0) {
1257         if (write_buffer (connptr->client_fd, connptr->sbuffer) < 0)
1258             break;
1259     }
1260     shutdown (connptr->client_fd, SHUT_WR);
1261
1262     /*
1263      * Try to send any remaining data to the server if we can.
1264      */
1265     ret = socket_blocking (connptr->server_fd);
1266     if (ret != 0) {
1267         log_message(LOG_ERR,
1268             "Failed to set server socket to blocking: %s",
1269             strerror(errno));
1270         return;
1271     }
1272
1273     while (buffer_size (connptr->cbuffer) > 0) {
1274         if (write_buffer (connptr->server_fd, connptr->cbuffer) < 0)
1275             break;
1276     }
1277
1278     return;
1279 }
1280
1281 static int
1282 connect_to_upstream_proxy(struct conn_s *connptr, struct request_s *request)
1283 {
1284     unsigned len;
1285     unsigned char buff[512]; /* won't use more than 7 + 255 */
1286     unsigned short port;
1287     size_t ulen, passlen;
1288
1289     struct upstream *cur_upstream = connptr->upstream_proxy;
1290
1291     ulen = cur_upstream->ua.user ? strlen(cur_upstream->ua.user) : 0;
1292     passlen = cur_upstream->pass ? strlen(cur_upstream->pass) : 0;
1293
1294
1295     log_message(LOG_CONN,
1296         "Established connection to %s proxy \"%s\" using file descriptor %d.",
1297         proxy_type_name(cur_upstream->type), cur_upstream->host, connptr->server_fd);
1298
1299     if (cur_upstream->type == PT_SOCKS4) {
1300
1301         buff[0] = 4; /* socks version */
1302         buff[1] = 1; /* connect command */
1303         port = htons(request->port);

```

```

1304     memcpy(&buff[2], &port, 2); /* dest port */
1305     memcpy(&buff[4], "\0\0\0\1" /* socks4a fake ip */
1306           "\0" /* user */, 5);
1307     len = strlen(request->host);
1308     if(len>255)
1309         return -1;
1310     memcpy(&buff[9], request->host, len+1);
1311     if (9+len+1 != safe_write(connptr->server_fd, buff, 9+len+1))
1312         return -1;
1313     if (8 != safe_read(connptr->server_fd, buff, 8))
1314         return -1;
1315     if (buff[0]!=0 || buff[1]!=90)
1316         return -1;
1317
1318     } else if (cur_upstream->type == PT_SOCKS5) {
1319
1320         /* init */
1321         int n_methods = ulen ? 2 : 1;
1322         buff[0] = 5; /* socks version */
1323         buff[1] = n_methods; /* number of methods */
1324         buff[2] = 0; /* no auth method */
1325         if (ulen) buff[3] = 2; /* auth method -> username / password */
1326         if (2+n_methods != safe_write(connptr->server_fd, buff, 2+n_methods))
1327             return -1;
1328         if (2 != safe_read(connptr->server_fd, buff, 2))
1329             return -1;
1330         if (buff[0] != 5 || (buff[1] != 0 && buff[1] != 2))
1331             return -1;
1332
1333         if (buff[1] == 2) {
1334             /* authentication */
1335             char in[2];
1336             char out[515];
1337             char *cur = out;
1338             size_t c;
1339             *cur++ = 1; /* version */
1340             c = ulen & 0xFF;
1341             *cur++ = c;
1342             memcpy(cur, cur_upstream->ua.user, c);
1343             cur += c;
1344             c = passlen & 0xFF;
1345             *cur++ = c;
1346             memcpy(cur, cur_upstream->pass, c);
1347             cur += c;
1348
1349             if((cur - out) != safe_write(connptr->server_fd, out, cur - out))
1350                 return -1;
1351
1352             if(2 != safe_read(connptr->server_fd, in, 2))

```

```

1353         return -1;
1354     if(in[1] != 0 || !(in[0] == 5 || in[0] == 1)) {
1355         return -1;
1356     }
1357 }
1358 /* connect */
1359 buff[0] = 5; /* socks version */
1360 buff[1] = 1; /* connect */
1361 buff[2] = 0; /* reserved */
1362 buff[3] = 3; /* domainname */
1363 len=strlen(request->host);
1364 if(len>255)
1365     return -1;
1366 buff[4] = len; /* length of domainname */
1367 memcpy(&buff[5], request->host, len); /* dest ip */
1368 port = htons(request->port);
1369 memcpy(&buff[5+len], &port, 2); /* dest port */
1370 if (7+len != safe_write(connptr->server_fd, buff, 7+len))
1371     return -1;
1372 if (4 != safe_read(connptr->server_fd, buff, 4))
1373     return -1;
1374 if (buff[0]!=5 || buff[1]!=0)
1375     return -1;
1376 switch(buff[3]) {
1377     case 1: len=4; break; /* ip v4 */
1378     case 4: len=16; break; /* ip v6 */
1379     case 3: /* domainname */
1380         if (1 != safe_read(connptr->server_fd, buff, 1))
1381             return -1;
1382         len = buff[0]; /* max = 255 */
1383         break;
1384     default: return -1;
1385 }
1386 if (2+len != safe_read(connptr->server_fd, buff, 2+len))
1387     return -1;
1388 } else {
1389     return -1;
1390 }
1391
1392 if (connptr->connect_method)
1393     return 0;
1394
1395 return establish_http_connection(connptr, request);
1396 }
1397
1398
1399 /*
1400  * Establish a connection to the upstream proxy server.
1401  */

```

```

1402 static int
1403 connect_to_upstream (struct conn_s *connptr, struct request_s *request)
1404 {
1405 #ifndef UPSTREAM_SUPPORT
1406     /*
1407      * This function does nothing if upstream support was not compiled
1408      * into tinyproxy.
1409      */
1410     return -1;
1411 #else
1412     char *combined_string;
1413     int len;
1414
1415     struct upstream *cur_upstream = connptr->upstream_proxy;
1416
1417     if (!cur_upstream) {
1418         log_message (LOG_WARNING,
1419                     "No upstream proxy defined for %s.",
1420                     request->host);
1421         indicate_http_error (connptr, 502,
1422                             "Unable to connect to upstream proxy.");
1423         return -1;
1424     }
1425
1426     connptr->server_fd =
1427         opensock (cur_upstream->host, cur_upstream->port,
1428                 connptr->server_ip_addr);
1429
1430     if (connptr->server_fd < 0) {
1431         log_message (LOG_WARNING,
1432                     "Could not connect to upstream proxy.");
1433         indicate_http_error (connptr, 502,
1434                             "Unable to connect to upstream proxy",
1435                             "detail",
1436                             "A network error occurred while trying to "
1437                             "connect to the upstream web proxy.",
1438                             NULL);
1439         return -1;
1440     }
1441
1442     if (cur_upstream->type != PT_HTTP)
1443         return connect_to_upstream_proxy(connptr, request);
1444
1445     log_message (LOG_CONN,
1446                 "Established connection to upstream proxy \"%s\" "
1447                 "using file descriptor %d.",
1448                 cur_upstream->host, connptr->server_fd);
1449
1450     /*

```

```

1451     * We need to re-write the "path" part of the request so that we
1452     * can reuse the establish_http_connection() function. It expects a
1453     * method and path.
1454     */
1455     if (connptr->connect_method) {
1456         len = strlen (request->host) + 7;
1457
1458         combined_string = (char *) safemalloc (len);
1459         if (!combined_string) {
1460             return -1;
1461         }
1462
1463         snprintf (combined_string, len, "%s:%d", request->host,
1464                 request->port);
1465     } else {
1466         len = strlen (request->host) + strlen (request->path) + 14;
1467         combined_string = (char *) safemalloc (len);
1468         if (!combined_string) {
1469             return -1;
1470         }
1471
1472         snprintf (combined_string, len, "http://%s:%d%s", request->host,
1473                 request->port, request->path);
1474     }
1475
1476     if (request->path)
1477         safefree (request->path);
1478     request->path = combined_string;
1479
1480     return establish_http_connection (connptr, request);
1481 #endif
1482 }
1483
1484 /* this function "drains" remaining bytes in the read pipe from
1485    the client. it's usually only called on error before displaying
1486    an error code/page. */
1487 static int
1488 get_request_entity(struct conn_s *connptr)
1489 {
1490     int ret;
1491     pollfd_struct fds[1] = {0};
1492
1493     fds[0].fd = connptr->client_fd;
1494     fds[0].events |= MYPOLL_READ;
1495
1496     ret = mypoll(fds, 1, config->idletimeout);
1497
1498     if (ret == -1) {
1499         log_message (LOG_ERR,

```

```

1500         "Error calling " SELECT_OR_POLL " on client fd %d: %s",
1501         connptr->client_fd, strerror(errno));
1502     } else if (ret == 0) {
1503         log_message (LOG_INFO, "no entity");
1504     } else if (ret == 1 && (fds[0].revents & MYPOLL_READ)) {
1505         ssize_t nread;
1506         nread = read_buffer (connptr->client_fd, connptr->cbuffer);
1507         if (nread < 0) {
1508             log_message (LOG_ERR,
1509                 "Error reading readable client_fd %d (%s)",
1510                 connptr->client_fd, strerror(errno));
1511             ret = -1;
1512         } else {
1513             log_message (LOG_INFO,
1514                 "Read request entity of %ld bytes",
1515                 (long) nread);
1516             ret = 0;
1517         }
1518     } else {
1519         log_message (LOG_ERR, "strange situation after " SELECT_OR_POLL ": "
1520             "ret = %d, but client_fd (%d) is not readable...",
1521             ret, connptr->client_fd);
1522         ret = -1;
1523     }
1524
1525     return ret;
1526 }
1527
1528 static void handle_connection_failure(struct conn_s *connptr, int got_headers)
1529 {
1530     /*
1531      * First, get the body if there is one.
1532      * If we don't read all there is from the socket first,
1533      * it is still marked for reading and we won't be able
1534      * to send our data properly.
1535      */
1536     if (!got_headers && get_request_entity (connptr) < 0) {
1537         log_message (LOG_WARNING,
1538             "Could not retrieve request entity");
1539         indicate_http_error (connptr, 400, "Bad Request",
1540             "detail",
1541             "Could not retrieve the request entity "
1542             "the client.", NULL);
1543         update_stats (STAT_BADCONN);
1544     }
1545
1546     if (connptr->error_variables) {
1547         send_http_error_message (connptr);
1548     } else if (connptr->show_stats) {

```



```

1549         showstats (connptr);
1550     }
1551 }
1552
1553
1554 /*
1555  * This is the main drive for each connection. As you can tell, for the
1556  * first few steps we are using a blocking socket. If you remember the
1557  * older tinypoxy code, this use to be a very confusing state machine.
1558  * Well, no more! :) The sockets are only switched into nonblocking mode
1559  * when we start the relay portion. This makes most of the original
1560  * tinypoxy code, which was confusing, redundant. Hail progress.
1561  *     - rjkaes
1562
1563  * this function is called directly from child_thread() with the newly
1564  * received fd from accept().
1565  */
1566 void handle_connection (struct conn_s *connptr, union sockaddr_union* addr)
1567 {
1568
1569     #define HC_FAIL() \
1570         do {handle_connection_failure(connptr, got_headers); goto done;} \
1571         while(0)
1572
1573     int got_headers = 0, fd = connptr->client_fd;
1574     size_t i;
1575     struct request_s *request = NULL;
1576     orderedmap hashofheaders = NULL;
1577
1578     char sock_ipaddr[IP_LENGTH];
1579     char peer_ipaddr[IP_LENGTH];
1580
1581     getpeer_information (addr, peer_ipaddr, sizeof(peer_ipaddr));
1582
1583     if (config->bindsame)
1584         getsock_ip (fd, sock_ipaddr);
1585
1586     log_message (LOG_CONN, config->bindsame ?
1587         "Connect (file descriptor %d): %s at [%s]" :
1588         "Connect (file descriptor %d): %s",
1589         fd, peer_ipaddr, sock_ipaddr);
1590
1591     if(!conn_init_contents (connptr, peer_ipaddr,
1592         config->bindsame ? sock_ipaddr : NULL)) {
1593         close (fd);
1594         return;
1595     }
1596
1597     set_socket_timeout(fd);

```

```

1598
1599     if (connection_loops (addr)) {
1600         log_message (LOG_CONN,
1601             "Prevented endless loop (file descriptor %d): %s",
1602             fd, peer_ipaddr);
1603
1604         indicate_http_error(connptr, 400, "Bad Request",
1605             "detail",
1606             "You tried to connect to the "
1607             "machine the proxy is running on",
1608             NULL);
1609         HC_FAIL();
1610     }
1611
1612
1613     if (check_acl (peer_ipaddr, addr, config->access_list) <= 0) {
1614         update_stats (STAT_DENIED);
1615         indicate_http_error (connptr, 403, "Access denied",
1616             "detail",
1617             "The administrator of this proxy has not configured "
1618             "it to service requests from your host.",
1619             NULL);
1620         HC_FAIL();
1621     }
1622
1623     if (read_request_line (connptr) < 0) {
1624         update_stats (STAT_BADCONN);
1625         goto done;
1626     }
1627
1628     /*
1629     * The "hashofheaders" store the client's headers.
1630     */
1631     hashofheaders = orderedmap_create (HEADER_BUCKETS);
1632     if (hashofheaders == NULL) {
1633         update_stats (STAT_BADCONN);
1634         indicate_http_error (connptr, 503, "Internal error",
1635             "detail",
1636             "An internal server error occurred while processing "
1637             "your request. Please contact the administrator.",
1638             NULL);
1639         HC_FAIL();
1640     }
1641
1642     /*
1643     * Get all the headers from the client in a big hash.
1644     */
1645     if (get_all_headers (connptr->client_fd, hashofheaders) < 0) {
1646         log_message (LOG_WARNING,

```

```

1647         "Could not retrieve all the headers from the client");
1648     indicate_http_error (connptr, 400, "Bad Request",
1649         "detail",
1650         "Could not retrieve all the headers from "
1651         "the client.", NULL);
1652     update_stats (STAT_BADCONN);
1653     HC_FAIL();
1654 }
1655 got_headers = 1;
1656
1657 if (config->basicauth_list != NULL) {
1658     char *authstring;
1659     int failure = 1, stathost_connect = 0;
1660     authstring = orderedmap_find (hashofheaders, "proxy-authorization");
1661
1662     if (!authstring && config->stathost) {
1663         authstring = orderedmap_find (hashofheaders, "host");
1664         if (authstring && !strncmp(authstring, config->stathost, strlen(config->stathost)))
1665             authstring = orderedmap_find (hashofheaders, "authorization");
1666         stathost_connect = 1;
1667     } else authstring = 0;
1668 }
1669
1670 if (!authstring) {
1671     if (stathost_connect) goto e401;
1672     update_stats (STAT_DENIED);
1673     indicate_http_error (connptr, 407, "Proxy Authentication Required",
1674         "detail",
1675         "This proxy requires authentication.",
1676         NULL);
1677     HC_FAIL();
1678 }
1679 if ( /* currently only "basic" auth supported */
1680     (strncmp(authstring, "Basic ", 6) == 0 ||
1681     strncmp(authstring, "basic ", 6) == 0) &&
1682     basicauth_check (config->basicauth_list, authstring + 6) == 1)
1683     failure = 0;
1684 if(failure) {
1685 e401:
1686     update_stats (STAT_DENIED);
1687     indicate_http_error (connptr, 401, "Unauthorized",
1688         "detail",
1689         "The administrator of this proxy has not configured "
1690         "it to service requests from you.",
1691         NULL);
1692     HC_FAIL();
1693 }
1694 orderedmap_remove (hashofheaders, "proxy-authorization");
1695 }

```

```

1696
1697 /*
1698  * Add any user-specified headers (AddHeader directive) to the
1699  * outgoing HTTP request.
1700  */
1701 if (config->add_headers)
1702 for (i = 0; i < sblist_getsize (config->add_headers); i++) {
1703     http_header_t *header = sblist_get (config->add_headers, i);
1704
1705     orderedmap_append (hashofheaders, header->name, header->value);
1706 }
1707
1708 request = process_request (connptr, hashofheaders);
1709 if (!request) {
1710     if (!connptr->show_stats) {
1711         update_stats (STAT_BADCONN);
1712     }
1713     HC_FAIL();
1714 }
1715
1716 connptr->upstream_proxy = UPSTREAM_HOST (request->host);
1717 if (connptr->upstream_proxy != NULL) {
1718     if (connect_to_upstream (connptr, request) < 0) {
1719         HC_FAIL();
1720     }
1721 } else {
1722     connptr->server_fd = opensock (request->host, request->port,
1723                                   connptr->server_ip_addr);
1724     if (connptr->server_fd < 0) {
1725         indicate_http_error (connptr, 500, "Unable to connect",
1726                               "detail",
1727                               PACKAGE_NAME " "
1728                               "was unable to connect to the remote web server.",
1729                               "error", strerror (errno), NULL);
1730         HC_FAIL();
1731     }
1732
1733     log_message (LOG_CONN,
1734                 "Established connection to host \"%s\" using "
1735                 "file descriptor %d.", request->host,
1736                 connptr->server_fd);
1737
1738     if (!connptr->connect_method)
1739         establish_http_connection (connptr, request);
1740 }
1741
1742 if (process_client_headers (connptr, hashofheaders) < 0) {
1743     update_stats (STAT_BADCONN);
1744     log_message (LOG_INFO,

```

```

1745         "process_client_headers failed: %s. host \"%s\" using "
1746         "file descriptor %d.", strerror(errno),
1747         request->host,
1748         connptr->server_fd);
1749
1750         HC_FAIL();
1751     }
1752
1753     if (!connptr->connect_method || UPSTREAM_IS_HTTP(connptr)) {
1754         if (process_server_headers (connptr) < 0) {
1755             update_stats (STAT_BADCONN);
1756             log_message (LOG_INFO,
1757                 "process_server_headers failed: %s. host \"%s\" using "
1758                 "file descriptor %d.", strerror(errno),
1759                 request->host,
1760                 connptr->server_fd);
1761
1762             HC_FAIL();
1763         }
1764     } else {
1765         if (send_connect_method_response (connptr) < 0) {
1766             log_message (LOG_ERR,
1767                 "handle_connection: Could not send CONNECT"
1768                 " method greeting to client.");
1769             update_stats (STAT_BADCONN);
1770             HC_FAIL();
1771         }
1772     }
1773
1774     relay_connection (connptr);
1775
1776     log_message (LOG_INFO,
1777         "Closed connection between local client (fd:%d) "
1778         "and remote client (fd:%d)",
1779         connptr->client_fd, connptr->server_fd);
1780
1781 done:
1782     free_request_struct (request);
1783     orderedmap_destroy (hashofheaders);
1784     conn_destroy_contents (connptr);
1785     return;
1786 #undef HC_FAIL
1787 }

```