```
/*
MicroSocks - multithreaded, small, efficient SOCKS5 server.

Copyright (C) 2017 rofl0r.

This is the successor of "rocksocks5", and it was written with
different goals in mind:

- prefer usage of standard libc functions over homegrown ones
- no artificial limits
- do not aim for minimal binary size, but for minimal source code size,
and maximal readability, reusability, and extensibility.

as a result of that, ipv4, dns, and ipv6 is supported out of the box
and can use the same code, while rocksocks5 has several compile time
defines to bring down the size of the resulting binary to extreme values
like 10 KB static linked when only ipv4 support is enabled.

still, if optimized for size, *this* program when static linked against musl
libc is not even 50 KB. that's easily usable even on the cheapest routers.

*/

#define _GNU_SOURCE
#include <unistd.h>
#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <poll.h>
#include <arpa/inet.h>
#include <errno.h>
#include <limits.h>
#include "server.h"
#include "sblist.h"

#ifndef MAX
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#endif

#ifdef PTHREAD_STACK_MIN
#define THREAD_STACK_SIZE MAX(8*1024, PTHREAD_STACK_MIN)
#else
#define THREAD_STACK_SIZE 64*1024
#endif

#if defined(__APPLE__)
#undef THREAD_STACK_SIZE
#define THREAD_STACK_SIZE 64*1024
#elif defined(__GLIBC__) || defined(__FreeBSD__)
#undef THREAD_STACK_SIZE
#define THREAD_STACK_SIZE 32*1024
```

```
#endif

static const char* auth_user;
static const char* auth_pass;
static sblist* auth_ips;
static pthread_rwlock_t auth_ips_lock = PTHREAD_RWLOCK_INITIALIZER;
static const struct server* server;
static union sockaddr_union bind_addr = {.v4.sin_family = AF_UNSPEC};
static int aimode;

enum socksstate {
SS_1_CONNECTED,
SS_2_NEED_AUTH, /* skipped if NO_AUTH method supported */
SS_3_AUTHED,
};

enum authmethod {
AM_NO_AUTH = 0,
AM_GSSAPI = 1,
AM_USERNAME = 2,
AM_INVALID = 0xFF
};

enum errorcode {
EC_SUCCESS = 0,
EC_GENERAL_FAILURE = 1,
EC_NOT_ALLOWED = 2,
EC_NET_UNREACHABLE = 3,
EC_HOST_UNREACHABLE = 4,
EC_CONN_REFUSED = 5,
EC_TTL_EXPIRED = 6,
EC_COMMAND_NOT_SUPPORTED = 7,
EC_ADDRESSTYPE_NOT_SUPPORTED = 8,
};

struct thread {
pthread_t pt;
struct client client;
enum socksstate state;
volatile int  done;
};

#ifndef CONFIG_LOG
#define CONFIG_LOG 1
#endif
#if CONFIG_LOG
/* we log to stderr because it's not using line buffering, i.e. malloc which would need
locking when called from different threads. for the same reason we use dprintf,
which writes directly to an fd. */
#define dolog(...) dprintf(2, __VA_ARGS__)
#else
static void dolog(const char* fmt, ...) { }
#endif
```

```
static int connect_socks_target(unsigned char *buf, size_t n, struct client *client) {
if(n < 5) return -EC_GENERAL_FAILURE;
if(buf[0] != 5) return -EC_GENERAL_FAILURE;
if(buf[1] != 1) return -EC_COMMAND_NOT_SUPPORTED; /* we support only CONNECT method */
if(buf[2] != 0) return -EC_GENERAL_FAILURE; /* malformed packet */

int af = AF_INET;
size_t minlen = 4 + 4 + 2, l;
char namebuf[256];
struct addrinfo* remote;

switch(buf[3]) {
case 4: /* ipv6 */
af = AF_INET6;
minlen = 4 + 2 + 16;
/* fall through */
case 1: /* ipv4 */
if(n < minlen) return -EC_GENERAL_FAILURE;
if(namebuf != inet_ntop(af, buf+4, namebuf, sizeof namebuf))
return -EC_GENERAL_FAILURE; /* malformed or too long addr */
break;
case 3: /* dns name */
l = buf[4];
minlen = 4 + 2 + l + 1;
if(n < 4 + 2 + l + 1) return -EC_GENERAL_FAILURE;
memcpy(namebuf, buf+4+1, l);
namebuf[l] = 0;
break;
default:
return -EC_ADDRESSTYPE_NOT_SUPPORTED;
}
unsigned short port;
port = (buf[minlen-2] << 8) | buf[minlen-1];
/* there's no suitable errorcode in rfc1928 for dns lookup failure */
if(resolve(namebuf, port, &remote, aimode)) return -EC_GENERAL_FAILURE;
int fd = socket(remote->ai_addr->sa_family, SOCK_STREAM, 0);
if(fd == -1) {
eval_errno:
if(fd != -1) close(fd);
freeaddrinfo(remote);
switch(errno) {
case ETIMEDOUT:
return -EC_TTL_EXPIRED;
case EPROTOTYPE:
case EPROTONOSUPPORT:
case EAFNOSUPPORT:
return -EC_ADDRESSTYPE_NOT_SUPPORTED;
case ECONNREFUSED:
return -EC_CONN_REFUSED;
case ENETDOWN:
case ENETUNREACH:
return -EC_NET_UNREACHABLE;
case EHOSTUNREACH:
return -EC_HOST_UNREACHABLE;
```

```
case EBADF:
default:
perror("socket/connect");
return -EC_GENERAL_FAILURE;
}
}
if(SOCKADDR_UNION_AF(&bind_addr) != AF_UNSPEC && bindtoip(fd, &bind_addr) == -1)
goto eval_errno;
if(connect(fd, remote->ai_addr, remote->ai_addrlen) == -1)
goto eval_errno;

freeaddrinfo(remote);
if(CONFIG_LOG) {
char clientname[256];
af = SOCKADDR_UNION_AF(&client->addr);
void *ipdata = SOCKADDR_UNION_ADDRESS(&client->addr);
inet_ntop(af, ipdata, clientname, sizeof clientname);
dolog("client[%d] %s: connected to %s:%d\n", client->fd, clientname, namebuf, port);
}
return fd;
}

static int is_authed(union sockaddr_union *client, union sockaddr_union *authedip) {
int af = SOCKADDR_UNION_AF(authedip);
if(af == SOCKADDR_UNION_AF(client)) {
size_t cmpbytes = af == AF_INET ? 4 : 16;
void *cmp1 = SOCKADDR_UNION_ADDRESS(client);
void *cmp2 = SOCKADDR_UNION_ADDRESS(authedip);
if(!memcmp(cmp1, cmp2, cmpbytes)) return 1;
}
return 0;
}

static int is_in_authed_list(union sockaddr_union *caddr) {
size_t i;
for(i=0;i<sblist_getsize(auth_ips);i++)
if(is_authed(caddr, sblist_get(auth_ips, i)))
return 1;
return 0;
}

static void add_auth_ip(union sockaddr_union *caddr) {
sblist_add(auth_ips, caddr);
}

static enum authmethod check_auth_method(unsigned char *buf, size_t n, struct client*client) {
if(buf[0] != 5) return AM_INVALID;
size_t idx = 1;
if(idx >= n ) return AM_INVALID;
int n_methods = buf[idx];
idx++;
while(idx < n && n_methods > 0) {
if(buf[idx] == AM_NO_AUTH) {
if(!auth_user) return AM_NO_AUTH;
```

```
else if(auth_ips) {
int authed = 0;
if(pthread_rwlock_rdlock(&auth_ips_lock) == 0) {
authed = is_in_authed_list(&client->addr);
pthread_rwlock_unlock(&auth_ips_lock);
}
if(authed) return AM_NO_AUTH;
}
} else if(buf[idx] == AM_USERNAME) {
if(auth_user) return AM_USERNAME;
}
idx++;
n_methods--;
}
return AM_INVALID;
}

static void send_auth_response(int fd, int version, enum authmethod meth) {
unsigned char buf[2];
buf[0] = version;
buf[1] = meth;
write(fd, buf, 2);
}

static void send_error(int fd, enum errorcode ec) {
/* position 4 contains ATYP, the address type, which is the same as used in the connect
request. we're lazy and return always IPV4 address type in errors. */
char buf[10] = { 5, ec, 0, 1 /*AT_IPV4*/, 0,0,0,0, 0,0 };
write(fd, buf, 10);
}

static void copyloop(int fd1, int fd2) {
struct pollfd fds[2] = {
[0] = {.fd = fd1, .events = POLLIN},
[1] = {.fd = fd2, .events = POLLIN},
};

while(1) {
/* inactive connections are reaped after 15 min to free resources.
usually programs send keep-alive packets so this should only happen
when a connection is really unused. */
switch(poll(fds, 2, 60*15*1000)) {
case 0:
send_error(fd1, EC_TTL_EXPIRED);
return;
case -1:
if(errno == EINTR || errno == EAGAIN) continue;
else perror("poll");
return;
}
int infd = (fds[0].revents & POLLIN) ? fd1 : fd2;
int outfd = infd == fd2 ? fd1 : fd2;
char buf[1024];
ssize_t sent = 0, n = read(infd, buf, sizeof buf);
```

```
if(n <= 0) return;
while(sent < n) {
ssize_t m = write(outfd, buf+sent, n-sent);
if(m < 0) return;
sent += m;
}
}
}

static enum errorcode check_credentials(unsigned char* buf, size_t n) {
if(n < 5) return EC_GENERAL_FAILURE;
if(buf[0] != 1) return EC_GENERAL_FAILURE;
unsigned ulen, plen;
ulen=buf[1];
if(n < 2 + ulen + 2) return EC_GENERAL_FAILURE;
plen=buf[2+ulen];
if(n < 2 + ulen + 1 + plen) return EC_GENERAL_FAILURE;
char user[256], pass[256];
memcpy(user, buf+2, ulen);
memcpy(pass, buf+2+ulen+1, plen);
user[ulen] = 0;
pass[plen] = 0;
if(!strcmp(user, auth_user) && !strcmp(pass, auth_pass)) return EC_SUCCESS;
return EC_NOT_ALLOWED;
}

static void* clientthread(void *data) {
struct thread *t = data;
t->state = SS_1_CONNECTED;
unsigned char buf[1024];
ssize_t n;
int ret;
int remotefd = -1;
enum authmethod am;
while((n = recv(t->client.fd, buf, sizeof buf, 0)) > 0) {
switch(t->state) {
case SS_1_CONNECTED:
am = check_auth_method(buf, n, &t->client);
if(am == AM_NO_AUTH) t->state = SS_3_AUTHED;
else if (am == AM_USERNAME) t->state = SS_2_NEED_AUTH;
send_auth_response(t->client.fd, 5, am);
if(am == AM_INVALID) goto breakloop;
break;
case SS_2_NEED_AUTH:
ret = check_credentials(buf, n);
send_auth_response(t->client.fd, 1, ret);
if(ret != EC_SUCCESS)
goto breakloop;
t->state = SS_3_AUTHED;
if(auth_ips && !pthread_rwlock_wrlock(&auth_ips_lock)) {
if(!is_in_authed_list(&t->client.addr))
add_auth_ip(&t->client.addr);
pthread_rwlock_unlock(&auth_ips_lock);
}
```

```
break;
case SS_3_AUTHED:
ret = connect_socks_target(buf, n, &t->client);
if(ret < 0) {
send_error(t->client.fd, ret*-1);
goto breakloop;
}
remotefd = ret;
send_error(t->client.fd, EC_SUCCESS);
copyloop(t->client.fd, remotefd);
goto breakloop;

}
}
breakloop:

if(remotefd != -1)
close(remotefd);

close(t->client.fd);
t->done = 1;

return 0;
}

static void collect(sblist *threads) {
size_t i;
for(i=0;i<sblist_getsize(threads);) {
struct thread* thread = *((struct thread**)sblist_get(threads, i));
if(thread->done) {
pthread_join(thread->pt, 0);
sblist_delete(threads, i);
free(thread);
} else
i++;
}
}

static int usage(void) {
dprintf(2,
"MicroSocks SOCKS5 Server\n"
"----------------------\n"
"usage: microsocks -1 -i listenip -p port -u user -P password -b bindaddr\n"
"all arguments are optional.\n"
"by default listenip is 0.0.0.0 and port 1080.\n\n"
"option -b specifies which ip outgoing connections are bound to\n"
"option -1 activates auth_once mode: once a specific ip address\n"
"authed successfully with user/pass, it is added to a whitelist\n"
"and may use the proxy without auth.\n"
"this is handy for programs like firefox that don't support\n"
"user/pass auth. for it to work you'd basically make one connection\n"
"with another program that supports it, and then you can use firefox too.\n"
);
return 1;
```

```
}

/* prevent username and password from showing up in top. */
static void zero_arg(char *s) {
size_t i, l = strlen(s);
for(i=0;i<l;i++) s[i] = 0;
}

int main(int argc, char** argv) {
int ch;
const char *listenip = "0.0.0.0"::";
aimode = AF_UNSPEC;
unsigned port = 1080;
while((ch = getopt(argc, argv, ":m:1b:i:p:u:P:")) != -1) {
switch(ch) {
case '1':
auth_ips = sblist_new(sizeof(union sockaddr_union), 8);
break;
case 'b':
resolve_sa(optarg, 0, &bind_addr, &aimode);
break;
case 'u':
auth_user = strdup(optarg);
zero_arg(optarg);
break;
case 'P':
auth_pass = strdup(optarg);
zero_arg(optarg);
break;
case 'i':
listenip = optarg;
break;
case 'p':
port = atoi(optarg);
break;
//          case 'm':
//             if ( atoi(optarg) == 4) aimode = PF_INET;
//             if ( atoi(optarg) == 6) aimode = AF_INET6;
//             printf("GetAddressInfo Mode: IPv%d \n",  atoi(optarg) );
//             break;
case ':':
dprintf(2, "error: option -%c requires an operand\n", optopt);
/* fall through */
case '?':
return usage();
}
}
if((auth_user && !auth_pass) || (!auth_user && auth_pass)) {
dprintf(2, "error: user and pass must be used together\n");
return 1;
}
if(auth_ips && !auth_pass) {
dprintf(2, "error: auth-once option must be used together with user/pass\n");
return 1;
```

```
}

if ( aimode == AF_INET ) printf("ipV4 only for GetAddressInfo\n");
if ( aimode == AF_INET6 ) printf("ipV6 only for GetAddressInfo\n");
if ( aimode == AF_UNSPEC ) printf("ipv4 + ipv6 for GetAddressInfo\n");

signal(SIGPIPE, SIG_IGN);
struct server s;
sblist *threads = sblist_new(sizeof (struct thread*), 8);
if(server_setup(&s, listenip, port)) {
perror("server_setup");
return 1;
}
server = &s;

while(1) {
collect(threads);
struct client c;
struct thread *curr = malloc(sizeof (struct thread));
if(!curr) goto oom;
curr->done = 0;
if(server_waitclient(&s, &c)) continue;
curr->client = c;
if(!sblist_add(threads, &curr)) {
close(curr->client.fd);
free(curr);
oom:
dolog("rejecting connection due to OOM\n");
usleep(16); /* prevent 100% CPU usage in OOM situation */
continue;
}
pthread_attr_t *a = 0, attr;
if(pthread_attr_init(&attr) == 0) {
a = &attr;
pthread_attr_setstacksize(a, THREAD_STACK_SIZE);
}
if(pthread_create(&curr->pt, a, clientthread, curr) != 0)
dolog("pthread_create failed. OOM?\n");
if(a) pthread_attr_destroy(&attr);
}
}
```

Removed: 1
Added: 7