**Southern Alberta Institute of Technology**

Information Systems Security Final Capstone

ITSC310

Windows  
**Remote Exploitation**With C Programming

Nick Faber

2020

Table of Contents

**Executive Summary3**

Instructions

**Project Procedure4**

Configuration

Persistence

Fingerprinting5

Networking

Processes6

Challenges7

Future Recommendations

**Conclusion8**

**References9**

# Executive Summary

The main purpose of this project was to gain more familiarity with the C programming language and experience with Windows development. Before this project, I had zero experience developing for Windows or even programming within a Windows environment.

Conceptually, the project began as a spyware program that would capture keystrokes to a file for later exfiltration (via USB, email, etc.). Eventually, server code was written, and the malware became a backdoor. Being a backdoor required me to prioritize size, networking, and evasion.

The binary had to be as small as possible, meaning avoiding static linking wherever possible. Every dynamic library function used by the malware is defined *defines.h*. Dynamic linking also makes the malware more portable to future versions of Windows.

Adding networking capability was the next priority, along with encrypting all inbound and outbound communication. Once this was functional, other components could be implemented. The highlight component is the keylogger, which was unfortunately detected by antivirus.

Subverting Bitdefender Antivirus ($90/yr!) was worryingly easy. Simply changing the order of a code block sufficed. The malware was then made persistent and able to retrieve a list of processes, kill a process, and finally, execute arbitrary commands as system.

# **Instructions**

Be sure to uncomment the first line of *main* before sending to a victim (for hidden execution).   
The following instructions only apply to users, reference README.txt for development configuration.

The malware’s default configuration can be changed in *defines.h*:

* **XOR** The character used in the XOR cipher
* **TIMEOUT** Seconds to sleep between connection attempts to the C2 server
* **DELAY** Seconds to sleep between sending information to the C2 server
* **NETSIZE** This many bytes can be sent to the C2 server every DELAY second(s)
* **MEMSIZE** This many bytes can be allocated to store keys, processes, etc.
* **STARTUP\_NAME** The name of the batch script created in the user’s Startup folder

The server can issue 4 commands to the client. Each command is represented by a number, and subsequent text (after a space delimiter) is interpreted as a parameter. CMD(#) PARAMETER:

* **1** The client returns a list of the victim’s current processes
* **2** Pass a processes name as a parameter to kill that specific process
* **3** The server will wait for a specified number of keystrokes to be returned
* **4** The parameters passed to this command will be executed by system (cmd.exe)

# Project Procedure

**Configuration**

Setting up the environment was more complicated than expected. This involved installing GCC and the Windows SDK, along with configuring Visual Studio Code to successfully build and debug the workspace. It was simple to do so after finding the right documentation online, but Linux development is surely more intuitive. The procedure I used to configure my environment can be seen in README.txt and the .vscode folder.

Once the environment was configured, it was time to fingerprint the victim. This involved getting the username, the current system time, and the operating system version. To keep the code organized, this functionality was placed in a separate file, *fingerprint.*c.

At first, I had difficulties getting a multi-file program to compile. This was because I was ignorant of header guards and the issue of double inclusion. After fixing this the program successfully compiled, and the code could ultimately be more organized. Now, additional files like *defines.h* and *functions.h* could also be created to keep function declarations and definitions organized.

**Persistence**

The first notable function that was created is *getUsername* in *fingerprint.c*. To get the username, a local buffer is allocated and then passed to *GetUserNameA*[1]. Once the username was in the local buffer, a function was needed to copy it into the buffer that was passed by reference to this function. I wanted to avoid using library functions like *strncpy* *(string.h)*, because statically linking them would drastically increase the size of the binary. I opted to write my own utility functions instead. To store these functions, *util.c* was created*.*

The first utility function that was made, and the one responsible for copying a string from one address to another, is *scpy.* It simply takes a pointer to the address being copied to and the address being copied from. To begin with, *scpy* received two more parameters: an int representing the length of the string (like *strncpy*), and a Boolean that determined if the two strings should be concatenated. Eventually these parameters were removed, and a separate function became responsible for string concatenation *(don’t worry about its name)*. I did this because *scpy* was often wasting clock cycles and I learned that the less parameters a function takes, the more modular code becomes.

With the username obtained, it is then used to get persistence. The malware does this by creating a batch script in the victim’s Startup directory. First, it retrieves its current directory from *GetCurrentDirectory*[2]*,* and then writes its complete path to a file pointer in ‘C:\Users\[USERNAME]\Microsoft\Windows\Start Menu\Programs\Startup\’. In hindsight, an environment variable can be used here as an alternative to multiple calls to the concatenation function. Once persistence is established, a smiley face is appended to the end of the client tag, as seen in *init (main.c)*.

**Fingerprinting**

Next, the client’s current local time is added to the start of the tag using *getTime* in *fingerprint.c*. This function first calls *GetLocalTime*[3]*,* passing it a reference to a local *SYSTEMTIME*[4] structure that receives the information. With the system’s local time returned, two utility functions were needed to copy it into the buffer, *mcpy* and *ntos.* The first is used here to copy the time string into the buffer without including the null byte. The second converts strings into integers. I had string formatting issues in *getTime* due to leading zeros and null bytes, but this was solved with careful consideration of how many bytes were being added to the buffer.

Then, the victim’s operating system version would be appended. To do so, the *getOsVersion* function calls *GetVersionExA*[5]with a reference to a local *OSVERSIONINFOEXA*[6] structure. This didn’t work at first, but after closer reading of MSDN, I realized that I didn’t set the *dwOSVersionInfoSize* member. This is where I began understanding how crucial reading is before typing. *getOsVersion* then allocates some local buffers and uses the major and minor version numbers returned by *GetVersionExA* to fill them accordingly. When finished, this function was oddly reporting my Windows 10 machine as version 8.1. Consulting MSDN, I found this to be because *GetVersionExA* only reports accurate versioning information up to Windows 8.0 – so, my day was wasted. Close reading before typing is crucial.

I didn’t want to waste the code in *getOsVersion* because it is still useful for versions under 8.1. Instead, I decided to query the registry if *GetVersionExA* was returning version 8.1.In hindsight, querying the version from the registry would also work for previous versions of Windows, so code from *getOsVersion* is redundant and keeping it unnecessarily increases the binary size. To query the registry, *verifyVersion* gets an open handle to ‘HKLM\Software\Microsoft\Windows NT\CurrentVersion’ by calling *RegOpenKeyExA*[7]*. Then,* the open handle is passed to *queryRegString* which is responsible for calling *RegQueryValueExA*[8]to retrieve the ‘ProductName’ and ‘CurrentBuild’ subkey values.

With the victim identified, the next functionality I had in mind was the keylogger. *getKeystrokes* in *hake.c,* iterates over all possible virtual-key codes[9] and calls *GetAsyncKeyState*[10]to determine if one is pressed. Next, it uses *GetKeyState*[11]to specifically check shift and caps lock (modifiers)*.* Any pressed key is identified and then added to a buffer.Initially, all the logged keys were all saved to a file.

**Networking**

Now to get the file off the system. Relying on MSDN[12], I established a simple client/server connection to send/receive commands. To achieve networking functionality, *connectServer (net.c)* usesmany functions from the winsock2 library. Initially, every function was dynamically called from Ws2\_32.dll. Unfortunately, I failed to create a local *SOCKET* type and *SOCKADDR* structure, and this necessitated static linking. I’m still unsure as to why this happened, because locally defined *WSADATA*[13] and *SOCKADDR\_IN*[14] structures behaved normally. Anyways, the client calls *WSAStartup*[15]to fill the *WSADATA* structure with information about the Windows Sockets implementation. The *SOCKADDR\_IN* structure is then filled with information about the C2 server’s configuration (protocol, address, port). The malware then binds an IPv4 TCP socket using the *socket*[16]function, and the connection is established after a call to *connect*[17].

In a loop, the client updates its timestamp, encrypts the buffer, and calls *send*[18]. It will then receive a command (using *recv*[19]), decrypt it, and pass it to the *exec* functionfor interpretation. This is a pattern that was often changed throughout the project and was the source of many bugs. Here, the file pointer used by *getKeystrokes* was replacedwith a buffer as well (allocated by *exec*)*.* The keystrokes are written to this buffer, and when *getKeystrokes* returns, the entirety of the buffer (sized MEMSIZE) is sent.

A new problem was created when MEMSIZE was greater than NETSIZE because the number of bytes being sent/received needed to be synchronized on both sides of the connection. To solve this, *sendbuffer* was created to send NETSIZE bytes of the buffer consecutively until empty, and the server logic was changed to accommodate these large client messages.

Now, MEMSIZE keystrokes could be stored in memory and then sent to the server in chunks of NETSIZE bytes. The previous sentence seems trivial, but wrestling with buffers to ensure their message’s integrity and avoid overflows absorbed many hours. This was especially apparent when implementing encryption (*cipher* in *util.c*). Eventually, I relied on a debugger to diagnose the buffers and solve the errors in my logic. In this case, the problems were often attributed to my utility functions and how null bytes were being considered – both things I would not have realized without GDB.

While bug hunting, I became impatient with being forced to wait for *getKeystrokes* to capture MEMSIZE keystrokes to be captured each time. NUM\_KEY was not yet undefined, and instead, the keylogging function filled *exec’s* buffer entirely. Ideally, the server should have been capable of telling the client how many keystrokes to grab. To do this, the interpreter (*exec*) was improved to support parameters being sent along with commands. Additionally, the inclusion of parameters allowed the client to execute any arbitrary commands with a simple call to *System*[20].

**Processes**

Being finished with the primary functionality, I then added the capability to exfiltrate a list of processes. In *hake.c, getProcesses* is passed a reference to the MEMSIZE buffer which gets filled with this list. To do so, *getProcesses* gets an open handle to a snapshot of the system by calling *CreateToolhelp32Snapshot*[21]. *Process32First*[22]is then called to retrieve information about the first process in the snapshot. For this to succeed, a *PROCESSENTRY32*[23] structure would first need to be created with its dwSize member set to the size of the structure. Then, in a loop, the name is copied to the MEMSIZE buffer (along with a delimiter), and the *PROCESSENTRY32* structure is filled with information for the next process by calling *Process32Next*[24].

There wasn’t much fun in getting a list of processes without being able to kill them. *killProcess* solved this dilemma, and it operates very similarly to the *getProcesses* function. It is identical in the way it retrieves an open handle to a system snapshot (returned by *CreateToolhelp32Snapshot*) and loads the first process from it into a *PROCESSENTRY32* structure with a call to *Process32First*. Where this function differs is by calling *scmp (util.c)* to compare the name of each process encountered in the snapshot to the argument that was passed to *killProcess*. When the names match, an open handle to the process object is retrieved by calling *OpenProcess*[25]. To specify the requested access rights, *OpenProcess* is passed *PROCESS\_TERMINATE*[26] as a parameter, and if access is granted, the process is finally terminated after a call to *TerminateProcess*[27].

**Challenges**

Thinking my work was finished, I copied the executable to my desktop… and BitDefender Antivirus flagged it as ‘Generic.Malware.SL!’. I hadn’t realized that up until this point, my antivirus was configured to avoid scanning my development environment for malware. To determine what code was being detected, I methodically deleted blocks of it until discovering that *getKeystrokes* was the culprit. This made sense, as the keylogger code is publicly available[28]. Knowing that malware operates on digital signatures, I simply mixed up the order in which the virtual key codes were checked in *getKeystrokes* (which changed the malwares digital signature). My code was now undetected yet fully functional, and my eyes were opened to the false security that antivirus provides.

Antivirus evasion was the final hurdle, but it was simple in comparison to the many other challenges encountered during this project. I will reiterate them here.

Setting up the development environment took more time than I had anticipated. I found installing the Windows SDK, GCC, and configuring Visual Studio Code successfully to be a painstaking process.

The first challenge I faced related to actual code was caused by double inclusion of the header file contents. This was fixed by learning about and implementing header guards.

I was often ignorant of programming best practices. For example, I tended to overload a function with parameters and conditions rather than creating a new one. This led to issues when the utility functions were wasting clock cycles.

Additionally, utility functions caused bugs when manipulating buffers. I often have difficulties with conceptualizing buffers/pointers; however, this challenge was greatly lessened by using GDB.

**Future Recommendations**

Exclusively use static linking to reduce size. Ideally, even the address for functions *GetProcAddress*[29] and *GetModuleHandleA*[30] would be dynamically called to avoid the libloaderapi library.

The client should be as remotely configurable as possible. Currently, it is hindered by hardcoded variables such as MEMSIZE and NETSIZE. This was caused by a lack of project foresight.

When querying the username and establishing persistence, the client could use the %USERPROFILE% and %APPDATA% environment variables rather than multiple library and *scpy* calls. When querying the operating system version, avoid *GetVersionExA’s* liesand exclusively use info returned from the registry*.* Exfiltrating processes and killing them could be done within a single function to avoid multiple library calls. Consider querying the addresses of all necessary DLL procedures once and saving them.

A language like Python would be much better suited for server code. The server’s binary size doesn’t matter, and Python’s flexibility would client handling and adding functionality much more bearable.

Document the code throughout the project. Commenting code while the MSDN document was open – and the logic was fresh in my mind – would have been much less time consuming.

# Conclusion

Programming is a very iterative process, and a project that begins as one thing can quickly evolve into another. For this reason, it is crucial to have a well-defined idea of what you are creating before your hands even touch the keyboard. During this project, I often found myself more focused on learning C programming concepts rather than Windows development concepts because I didn’t have a well-defined end goal in mind.

Having a clear idea of the project outcome will improve the code’s functionality and modularity. Because networking functionality was an afterthought in this project, the client was ultimately hindered by statically defined parameters (e.g. NETSIZE and MEMSIZE). Additionally, code that was added as an afterthought consistently caused bugs. This was most notably seen when the buffers of newer functions were being manipulated by utility functions that were developed without future networking capabilities in mind.

To diagnose and fix broken code, using a debugger is crucial. By the end of this project, I was even using GDB to step through freshly written code. Doing this improved my conceptualization and highlighted logical inconsistencies before they evolved into bugs.

A developer can further comprehend their code – and save time – with consistent documentation. I neglected to do this and suffered as a result. This project’s first comments were written in the final months, and the documentation process took much more time and effort for this reason.

Fortunately, the time that was lost to writing documentation was made up for when antivirus was subverted much more quickly than I had anticipated. For this reason, I can certainly conclude that antivirus is useless against targeted malware. Any malware can be easily made undetectable if the author can mimic the victim’s environment and test the virus’ detection throughout development.

Malware development is not difficult, it is time consuming. Before this project, I abstained from starting programming projects on my own accord because I was intimidated by their perceived difficulty. Now I realize that development only appears to be difficult due to the intimidating amount of API documentation and tutorials one must digest for success. Any patient person with a willingness to read and a basic understanding of language syntax can be a successful developer.

What this project has ultimately taught me is that you will never know everything about the language/environment you are working with, the platform you are developing for, or even the project itself. Therefore, a recognition of this coupled with patience and a desire to learn facilitates project success and personal growth.

reference the included code and documentation for additional insight into this project

# References

[1] GetUserNameA function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getusernamea>

[2] GetCurrentDirectory function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getcurrentdirectory>

[3] GetLocalTime function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getlocaltime>

[4] SYSTEMTIME structure. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-systemtime>

[5] GetVersionExA function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getversionexa>

[6] OSVERSIONINFOEXA structure. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-osversioninfoexa>

[7] RegOpenKeyExA function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regopenkeyexa>

[8] RegQueryValueExA function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regqueryvalueexa>

[9] Virtual-Key Codes. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes>

[10] GetAsyncKeyState function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getasynckeystate>

[11] GetKeyState function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getkeystate>

[12] Windows server code. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/winsock/complete-server-code>

[13] WSADATA structure. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winsock/ns-winsock-wsadata>

[14] SOCKADDR\_IN structure. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/ws2def/ns-ws2def-sockaddr_in>

[15] WSAStartup function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup>

[16] socket function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-socket>

[17] connect function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-connect>

[18] send function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-send>

[19] recv function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock-recv>

[20] System function. Retrieved from <https://docs.microsoft.com/en-us/cpp/c-language/system-function?view=vs-2019>

[21] CreateToolhelp32Snapshot function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-createtoolhelp32snapshot>

[22] Process32First function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-process32first>

[23] PROCESSENTRY32 structure. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/ns-tlhelp32-processentry32>

[24] Process32Next function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-process32next>

[25] OpenProcess function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

[26] Process Security and Access Rights. <https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights>

[27] TerminateProcess function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess>

[28] Windows keylogger code. Retrieved from <https://packetstormsecurity.com/files/150023/Windows-x64-Remote-Bind-TCP-Keylogger-Shellcode.html>

[29] GetProcAddress function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>

[30] GetModulHandleA function. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea>