Hijacking an embedded, statically linked python interpreter in an external process under win32

Marc Förster
Institute for Distributed Systems
Ostfalia Hochschule für Angewandte Wissenschaften
Wolfenbüttel, Deutschland
mail@lpcvoid.com

***Abstract*—This document describes an attack on an external, embedded python interpreter running in a separate process on win32. It enables the attacking process to force the target process to execute arbitrary Python code within the main interpreter context.**

***Index Terms*—win32, code injection, python**

I. Introduction

Python [2] is a general purpose programming language, which is commonly embedded into other applications for the purpose of adding scripting capabilities. Embedded in this case means that the target application has no dependency to any external DLL, as the python library was statically linked into the application executable at compiletime.

The attack is performed by injecting x86 code into the target address space, which will in turn call the python interpreter with the code supplied by the attacking process.

II. Code injection

A. General

The term *code injection* can refer to many scenarios. Within the scope of this paper, it means that arbitrary machine code is written by one process (attacker) into another (target), and then executed to run within the targets address space. This technique is used by several types of programs, for example debuggers. They use this technique to alter the control flow of debugged targets, for example by writing breakpoints or changing code on the fly while debugging, if the user wishes so.

Programs that are written to cheat in online games also use some of the functions presented within this paper.

While most use cases are certainly without bad intentions, this technique can easily be abused for nefarious purposes [1]. One such purpose will be demonstrated in this paper.

B. Methods

On the win32 platform, there are several possible methods to inject code into a foreign process. Two of these shall be discussed here, but just one of them will be used throughout this paper. In technical terms, both methods do not differ much from each other. Both rely on allocating space inside the target, writing data and/or code, and executing that code within the target. Depending on the method used, some steps may be omitted.

1) Process access: Gaining access to the process is the first thing an attacker has to do in both of these methods. It requires Administrator privileges, because the attacker needs to call the *OpenProcess* [7] function with *PROCESS_VM_OPERATION*, *PROCESS_VM_WRITE* and *PROCESS_CREATE_THREAD* privileges in order to get a handle usable for further operation. Depending on the goal of the attacker, *PROCESS_VM_READ* may also be required. Windows offers a macro where all privileges are already set, named *PROCESS_ALL_ACCESS*. This flag is passed to *OpenProcess* using the *dwDesiredAccess* parameter. It also needs two other parameters, where only the last one, *dwProcessId*, is relevant in scope of this paper. The *dwProcessId* parameter requires the Process ID of the target process. For more information on process access rights, see [8].

```
HANDLE OpenProcess(
DWORD dwDesiredAccess,
BOOL  bInheritHandle,
DWORD dwProcessId
);
```

After the process handle is obtained with the correct privileges, the attacker can continue with the attack.

2) DLL injection: A very common approach to getting another process to execute external code within its own address space is via use of the Windows DLL loader. A DLL is a dynamic library containing auxiliary code on the Windows operating system. In technical terms, it requires that the path to the DLL (or the name, if the path is in the PATH environment variable) is located in memory. This is then passed as a parameter to *LoadLibrary* [3], a winapi function to dynamically load a DLL. It must be pushed to the stack before calling the api function, since it expects the filename as a string. In case the path (or name) of the DLL is not somewhere in the targets memory already, it is required to write it first. This is done using *WriteProcessMemory* [4] Api.

3) Direct code injection: The second method is direct code injection. Contrary to DLL injection, where the goal is to call *LoadLibraryA* or *LoadLibraryW* depending on Unicode constraints, with direct code injection the goal is to execute injected code directly. The advantage of this is that it's harder

[1] Many malware authors use this technique to trick security software.

to detect than DLL injection, and has a much smaller footprint under normal circumstances, since only a small fragment of code is written into the target application, instead of a whole DLL loaded at runtime. The downside is that normally direct code injection is more complex, since the attacker cannot always rely on the compiler just outputting a ready-to-inject binary.

C. Methology of direct code injection

1) Memory Allocation: Typically, the first step (after having aquired a process handle as described in II-B1 on the preceding page) of a direct code injection is finding or creating some memory where the code that is to be executed can be safely written and later on executed. Ideally, the attacker does not have to allocate any memory for this. Instead, he finds a region of memory inside the target address space that isn't populated by anything else. This often occurs between functions, where compilers insert padding. This region between two functions is often called a *code cave* [5].

```
83C4 04          add     esp, 0x4
8B45 D8          mov     eax, dword ptr ss:[ebp - 0x28]
8B4D F4          mov     ecx, dword ptr ss:[ebp - 0xC]
64:890D 00000000 mov     dword ptr fs:[0], ecx
8BE5             mov     esp, ebp
5D               pop     ebp
C3               ret
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
CC               int3
55               push    ebp
8BEC             mov     ebp, esp
81EC 58030000    sub     esp, 0x358
898D A8FCFFFF    mov     dword ptr ss:[ebp - 0x358], ecx
C745 F4 00000000 mov     dword ptr ss:[ebp - 0xC], 0x0
8B45 0C          mov     eax, dword ptr ss:[ebp + 0xC]
```

Fig. 1. Example of a code cave

The padding in this case consists of 0xCC bytes, which translate to an int3 instruction in x86 assembly. This means it will trigger interrupt 3, which is another term for a software breakpoint. Despite what some sources online say, *code cave* paddings do not always contain 0x00 bytes. This depends on the used compiler.

Within this region, an attacker is free to place whatever code he wants, as it is never used within normal operation of the target application. A glaringly obvious advantage of using a code cave is that no memory needs to be allocated. Also, since these code caves reside in memory pages that are already set to be executable due to the actual program code residing in them, marking the page as executable is not required. A downside to this technique is that these code caves are often times too small to use for anything more substantial, as they rarely reach adequate size. For simple trampoline attacks, where the attackers injected code simply calls a function within the targets address space, it may be enough. But for everything else, where code sizes can approach multiple KiB, memory needs to be allocated. Another disadvantage is that these *code caves* are normally not at a known address. They need to be searched for, which can take considerable time and resources an attacker may not have. Due to the disadvantages of *code caves*, specially in regard to their small size, code will be allocated by the attacker within the targets address space for the purposes of this paper. This is done using the winapi function *VirtualAllocEx* [6].

```
LPVOID WINAPI VirtualAllocEx(
_In_     HANDLE hProcess,
_In_opt_ LPVOID lpAddress,
_In_     SIZE_T dwSize,
_In_     DWORD  flAllocationType,
_In_     DWORD  flProtect
);
```

The first parameter *hProcess* is the process handle, which was obtained by *OpenProcess*. Following that is *lpAddress*, which is not interesting in this case, since the attacker does not care where the allocated memory resides within the process. *dwSize* is much more interesting, since the attack needs a certain amount of memory to write its code and parameters. *flAllocationType* is also important, since the attacker wants pages to actually be allocated and ready to use. The last parameter, *flProtect*, defines the memory protection options for the newly allocated memory. This parameter depends on what shall be done with the memory later on. The attack demonstrated within this paper requires code execution capabilities along with normal data I/O, which is why it is requested that this region is readable, writable and executable. For sake of simplicity, two regions are allocated. One for the x86 code, and one for the parameters that this code needs (which, along with other data, also includes the python script that is to be executed as the primary goal of this attack).

Target	Size	Memory protection
x86 code	constant	*PAGE_EXECUTE_READWRITE*
parameters	sizeof(parameters)	*PAGE_READWRITE*

TABLE I
DIFFERENT PARAMETERS OF PAGE ALLOCATION

Again, to be concise, one memory region would be enough, since both data and code could be written consecutively, as long as the x86 code is terminated (*ret* x86 instruction) properly. Using two regions is just more comfortable. It is up to the reader to decide if his scenario allows for this comfort.

2) Writing to allocated memory: Now that the attacker has memory within the target process, which he can freely write to, actually writing is the next step. Under windows, there are multiple ways to write to a foreign memory region. The one presented in this paper is a winapi function called *WriteProcessMemory*.

```
BOOL WINAPI WriteProcessMemory(
_In_  HANDLE  hProcess,
_In_  LPVOID  lpBaseAddress,
_In_  LPCVOID lpBuffer,
_In_  SIZE_T  nSize,
_Out_ SIZE_T  *lpNumberOfBytesWritten
);
```

hProcess is the process handle, *lpBaseAddress* is the address where data is to be written (as allocated by *VirtualAllocEx*

previously), *lpBuffer* is the pointer to a buffer that shall be written (located in attacker address space), *nSize* is the size of the buffer, and *lpNumberOfBytesWritten* points to a *long* that will contain the actual written byte count after execution. This parameter is important, since it lets the attacker know if all bytes where written correctly. Failure to do so would cause the target process to crash with high certainty.

3) Executing injected code: Providing that the attacker has sufficient rights, the attacker now owns a memory region within the target process, which contains both executable x86 code written by the attacker, and data. This x86 code can do whatever the attacker wants, which should instantly demonstrate the danger of this attack. The last step is executing this code within the target. The attacker can do this via multiple methods, but this paper concentrates on the most straight forward one, namely an api called *CreateRemoteThread* [9].

```
HANDLE CreateRemoteThread(
HANDLE                hProcess,
LPSECURITY_ATTRIBUTES  lpThreadAttributes,
SIZE_T                dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID                lpParameter,
DWORD                 dwCreationFlags,
LPDWORD               lpThreadId
);
```

This is a very powerful function, which allows a thread to be created in a foreign address space. The function expects, similar to the previous ones, a thread handle *hProcess*. The second parameter, *lpThreadAttributes*, is set to NULL since the attacker does not care to pass a security descriptor [10] for this attack. Parameter *dwStackSize* describes the default stack size of the thread that is to be created. The attacker sets this value to 0, so that the default size is used, which is 1MiB on the MSVC compiler [2]. A lower value could be considered, depending on the complexity of the injected x86 code that shall run. The next two parameters, *lpStartAddress* and *lpParameter*, are pointers located in the address space where the thread is to be created (target). In this case, they are the addresses of the two buffers allocated by *VirtualAllocEx*. *dwCreationFlags* makes it possible to define flags that affect the behavior of the thread. Since the attacher does not want the thread to run immediately after creation, he passes *CREATE_SUSPENDED* here. Otherwise, the attacker could pass a 0 (no flags), which immediately starts execution. The reason the thread shall start in a suspended state is so the attacker can do other things before execution, such as making sure that the target is in a certain state, or cleanup operations. Once the attacker is ready for his injected code to execute, he calls *ResumeThread* [12] with the thread handle of the created thread, which was returned by *CreateRemoteThread*.

```
DWORD ResumeThread(
HANDLE hThread
);
```

[2]can be changed via *STACKSIZE* statement in module definition file

This sets the thread to running, and the code is executed. To make it possible to wait for the end of execution, the attacker should use *WaitForSingleObject* [11]. This is important since it allows the attacker to react to the end of execution, for when he wants to perform cleanup. Otherwise, the risk of premature cleanup is high, and doing so can potentially crash the target application.

```
DWORD WaitForSingleObject(
HANDLE hHandle,
DWORD  dwMilliseconds
);
```

This function expects two parameters. The first is *hHandle*, same as for *ResumeThread*. The second parameter, *dwMilliseconds*, specifies the amount of time that windows shall wait for the execution of the injected thread to end, in milliseconds. The attacker sets this to a value acceptable to him.

4) Cleaing up: After *WaitForSingleObject* returns, the thread has run successfully and executed the injected code. The attack was a success. Now the attacker can clean up the thread context using *CloseHandle* winapi call.

```
BOOL WINAPI CloseHandle(
_In_ HANDLE hObject
);
```

This function expects the handle of the thread. It can be used on any HANDLE object within Windows.

D. Naked code

The direct code injection method presented in II-C on the previous page makes it possible to execute code within a target process. This code needs to be carefully crafted. For this attack, the code shall call a function of the python interpreter that is used to execute Python code, along with two auxiliary functions which are Python specific. That means there are three addresses which have to be called, each one with different parameters.

Naked code is achieved by marking a function with the naked keyword. This forces the compiler to generate code without a *prolog* and *epilog*.

```
__declspec(naked) void static __stdcall
    inject_function() {}
```

Since the attacker is injecting this code into a remote address space, he cannot rely on any other external functions. Also, the attacker is responsible for stack maintenance. Additionally, removing compiler generated code that is often not needed for the injection to work reduces function size. This makes naked code an attractive choice.

prolog code is code that the compiler generates for a given function which sets up the stack in order for the function to execute properly. This includes local variables, function arguments, and the return address pushed to stack by the *call* opcode in x86. In x86 with *cdecl* calling convention, which is what most targets use, the *ebp* register contains a pointer

to the current stack frame, and this must be setup before the actual function code is set up.

```
push  ebp
mov   ebp, esp
sub   esp, 8 ; add stack space for local variables
```

epilog does the exact opposite, it restores the stack pointer, and returns to the caller.

```
mov   esp, ebp
pop   ebp
ret
```

These two elements will not be automatically generated when using naked code, and that's an attribute the attacker wants, since he should have as much granularity about the injected code as possible. For this reason, the code the attacker constructs will be marked naked, and the attacker must implement various steps himself.

III. Attacking Python

A. Overview

Python is a general purpose programming language with rising popularity [14]. Many programs chose to embed Python as a method to add scripting capabilities to an application. There are two methods to do this. The first one is to use a shared library [3], which gets loaded into the applications address space at runtime through the PE/ELF loader. The other method is compiling Python directly into the host application, a process referred to as "embedding". While the shared library approach is far easier to implement due to the ease of use that such a library offers, embedding the source code is a more complex task. This is because the set of function a shared library offers is normally designed for external use, where embedding a scripting language in general often times requires wrapping its interface in a more project related manner. Fortunately though for Python, there is a "high level" api exposed by the *pythonrun.h* header. It is also documented within the Python documentation [15].

B. Target function

Within *pythonrun.h*, there exists one function which is of particular interest for this paper, namely *PyRun_SimpleStringFlags*.

```
int PyRun_SimpleStringFlags(const char *command,
    PyCompilerFlags *flags)
```

This function accepts a char pointer to a buffer that contains Python source code, called *command*. The second parameter is not relevant under normal circumstances, but may be useful in special scenarios. This function is the one this attack aims to call within the target application, with the attackers code as argument.

[3] a DLL (Dynamic Link Library) under Windows, or an SO (Shared Object) under POSIX flavored operating systems

PyRun_SimpleStringFlags expects the Python subsystem to be properly initialized before being called. This initialization is not subject to this paper, but in most cases the reader does not need to worry about it anyhow, since it was most likely already called by the target application, as it needs to initialize the Python subsystem for itself at some point. For sake of being complete, *PyInitialize()* needs to be called. Additional calls need to be made in order to avoid crashes when executing this attack, but more about that later.

Obviously, this attack depends on this function being available somewhere within the targets application code. If the programmer of the target application removed it, this attack needs to be adjusted. The attacker could in that case emulate the behavior of *PyRun_SimpleStringFlags*, which is fairly trivial too, as the whole function is only 102 bytes long in Python 2.7. It calls a total of seven subfunctions, of which two could theoretically be omitted[4]. Alternatively, the attacker could also reverse engineer the target application to find the interface to the Python interpreter that the target uses itself, and adjust this attack even further. The author of this paper has never seen an embedded Python interpreter in the wild that attempted to hide the interface described in this paper though. Should the reader ever need to search for this function in a target application, he should use the following byte sequence, which can locate the function within any embedded interpreter.

```
83C40483C8FF5DC38301FF
```

This sequence locates the following x86 machine code, which is located in *PyRun_SimpleStringFlags* at offset 0x3C.

```
PyRun_SimpleStringFlags+0x3C add esp, 4
PyRun_SimpleStringFlags+0x3F or eax, 0xFFFFFFFF
PyRun_SimpleStringFlags+0x42 pop ebp
PyRun_SimpleStringFlags+0x43 retn
PyRun_SimpleStringFlags+0x44 add [ecx], 0xFFFFFFFF
```

C. Global interpreter lock

Unfortunately, the attacker cannot simply inject code that calls this function with two parameters, since *PyRun_SimpleStringFlags* is not thread safe. Due to the nature of this attack, where the attacker creates a thread within the address space of the target, he automatically makes the target application a multi threaded application, albeit for just a short period of time, namely while executing the attackers code. This can potentially crash the whole application, since the Python interpreter, amongst many other things, does not maintain reference count in a threadsafe fashion. For this reason, Python introduced the *Global Interpreter Lock*, in short *GIL*. It introduced a possibility to lock the interpreter so that several threads can consecutively use it after each other, without the danger of a crash. Using the GIL in software requires two functions. First, *PyGILState_Ensure* is called, which returns a *PyGILState_STATE* identifier, which is actually just an integer. This identifier needs to be stored, and afterwards Python code may be executed. When the

[4] PyErr_Print() and PyErr_Clear(), these are only error handling.

thread is done executing Python, it unlocks the interpreter again, so that another thread may use it. This is done via *PyGILState_Release*, which expects the *PyGILState_STATE* identifier as a parameter.

In order for this mechanism to work, the Python interpreter needs to be initialized with two additional calls, ideally directly after *PyInitialize* was called. These two functions are *PyEval_InitThreads* and *PyEval_ReleaseLock*. The first call initializes the GIL. It is important that this function is called before the GIL is obtained the first time with *PyGILState_Ensure*, otherwise the result is undetermined. The second call releases the GIL for use with other threads. Most applications that embed Python call these functions beforehand. If this is not the case though, the attacker must make sure that the application does do so, for example by hooking the function that sets up the Python interpreter, and forcing it to execute them additionally to the rest of the initialization.

D. Constructing x86 attack code

The code that is to be injected into the target process needs to fullfill several constraints. First and foremost, it must be able to execute Python code. This is done by acquiring the GIL using *PyGILState_Ensure*, saving the identifier it returns, calling *PyRun_SimpleStringFlags* to actually execute Python code, and then finally return the GIL with *PyGILState_Release*. The second constraint is that it should be small, so the amount of memory the attacker needs to allocate within the process remains small. Third, it needs to be parameterizable. This means that the code shouldn't need to be changed in order to execute it with different parameters. This is why the attacker should allocate a separate memory region for any data that could change. That includes the actual Python script, and any constants that the attacker could want to change. These constants are the addresses of the three functions he wants to call within the address space of the target process. The reason the attacker should not define these directly within the naked code, is that this technique enables him to later on update the addresses easily, without needing to recompile the naked function.

As a side note, it is, of course, possible to first write the naked code with placeholders into memory, and then patching these placeholders with the correct values before execution. This has the advantage of faster code execution, as addresses do not need to be copied out of the parameter memory. The downside in this case would be the need for at least one more external *WriteProcessMemory* call. All things considered, the author of this paper recommends keeping addresses together with the parameters.

The code, once started, isn't very different to any other function call. It follows the cdecl calling convention of pushing parameters to stack in reverse order, while stack cleanup and synchronization is done by the caller. Since all arguments are widened to 32 bit, each literal pushed, added or otherwise processed is a 32 bit integer. Results of functions are returned in the *eax* register as widened 32 bit integers. Note that these rules apply for 32 bit applications. For 64 bit, there are other rules to be considered, see [18]. The 64 bit ABI[5] also differs by data type.

The following is a proof-of-concept naked function that satisfies the above three criteria, and aims to be self explaining. This code makes minimal use of the stack (for parameters and for saving the GIL).

```
__declspec(naked) void static execute_py_str() {
  __asm {
  mov edx, [esp + 4]
  mov edx, [edx + 4]    //PyGILState_Ensure
  call edx
  push eax
  push 0
  mov edx, [esp + 0xC]
  add edx, 0xC
  push edx
  mov edx, [esp + 0x10]
  mov edx, [edx + 0]    //PyRun_SimpleStringFlags
  call edx
  add esp, 8h
  mov edx, [esp + 8]
  mov edx, [edx + 8]    //PyGILState_Release
  call edx
  add esp, 4h
  ret
  };
}
```

Note that the topmost address on the stack is the return address of the calling function. Since the attacker is not interested in it, he skips it in the first line and directly accesses the second address, which is a pointer to the second memory region the attacker allocated previously for the parameters. This parameter memory is structured as follows.

```
#define PYCODE_LEN 8000
struct memInjectParams {
  DWORD PyRun_SimpleStringFlags_addr;
  DWORD PyGILState_Ensure_addr;
  DWORD PyGILState_Release_addr;
  char python_code[PYCODE_LEN];
};
```

After the x86 code and the parameter struct are written to their respective memory locations, it becomes obvious that the code does not need to be changed again. Whenever the attacker needs to execute a different script, he simply writes the new Python code to *memInjectParams.python_code* and executes the injected code again using *CreateRemoteThread*.

IV. Countermeasures

Within the past years, software security requirements have increased. While this attack does not exploit any software security bugs, it can nontheless compromise a running Python application, in the worst case without the operator knowing about it. The attack in this paper can be performed by anyone, and requires no special software other than a compiler. It does, however, require that the attacking application is given elevated privileges, such as being started as Administrator. Note that under Windows XP, this is not a requirement, which

[5] Aplication Binary Interface

should further underline the need to migrate to a more secure version of the operating system. Added to that, programmers seeking to make this technique more complicated for the attacker should remove every part of the Python library that's not used internally, as this reduces the possibilities of attack, since the hardest code to attack is the one that is missing.

If the attacker uses DLL injection as presented in this paper for deploying attack code into a target process, it is also possible to detect due to the *DLL_THREAD_ATTACH* messages sent if the process is debugging itself. Furthermore, even direct code injection can be detected using a TLS[6] callback, which makes it possible for the target application to be notified when threads are created within its address space. Of course, now the attacker could patch this function, starting a game of cat and mouse.

Methods such as enumerating all loaded modules to search for unknown DLLs are quickly defeated and should not be used, especially considering that there exist many legitimate software that injects DLLs into windows processes. Window managers come to mind, along with theming engines and even some screen capturing software.

Hashing memory regions to detect injected x86 code is a valid approach which a lot of anti-cheat protections for games apply, such as Xtrap [16] and GameGuard [17], though these protection schemes offer much more complex protection than this.

In general, countermeasures for the discussed technique would warrant writing a separate paper, complexity wise.

V. Conclusion

The technique presented in this paper allows an attacking process to force an external python interpreter to execute the attackers code. This was done via x86 code injection, where x86 code that called relevant Python functions was injected into the target process. Countermeasures where discussed briefly towards the end.

References

[1] Microsoft. (2019) naked (C++) — Microsoft Docs. Retrieved January 08, 2019, from https://docs.microsoft.com/en-us/cpp/cpp/naked-cpp?view=vs-2017

[2] Python Foundation. (2019) https://www.python.org/. Retrieved January 08, 2019, from https://www.python.org/

[3] Microsoft. (2019) LoadLibraryA function — Microsoft Docs. Retrieved January 08, 2019, from https://docs.microsoft.com/en-us/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibrarya

[4] Microsoft. (2019) WriteProcessMemory function (Windows). Retrieved January 08, 2019, from https://msdn.microsoft.com/en-us/library/windows/desktop/ms681674(v=vs.85).aspx

[5] Code Cave. (2019) code cave - Wiktionary. Retrieved January 08, 2019, from https://en.wiktionary.org/wiki/code_cave

[6] Microsoft. (2019) VirtualAllocEx function (Windows). Retrieved January 08, 2019, from https://msdn.microsoft.com/en-us/library/windows/desktop/aa366890(v=vs.85).aspx

[7] Microsoft. (2019) OpenProcess function — Microsoft Docs. Retrieved January 08, 2019, from https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess

[8] Microsoft. (2019) Process Security and Access Rights - Windows applications — Microsoft Docs. Retrieved January 08, 2019, from https://docs.microsoft.com/en-us/windows/desktop/ProcThread/process-security-and-access-rights

[9] Microsoft. (2019) CreateRemoteThread function — Microsoft Docs. Retrieved January 09, 2019, from https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread

[10] Microsoft. (2019) SECURITY_ATTRIBUTES structure (Windows). Retrieved January 09, 2019, from https://msdn.microsoft.com/en-us/56b5b350-f4b7-47af-b5f8-6a35f32c1009

[11] Microsoft. (2019) WaitForSingleObject function — Microsoft Docs. Retrieved January 09, 2019, from https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject

[12] Microsoft. (2019) ResumeThread function — Microsoft Docs. Retrieved January 09, 2019, from https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread

[13] Microsoft. (2019) CloseHandle function (Windows). Retrieved January 09, 2019, from https://msdn.microsoft.com/en-us/library/windows/desktop/ms724211(v=vs.85).aspx

[14] TIOBE. (2019) Python — TIOBE - The Software Quality Company. Retrieved January 09, 2019, from https://www.tiobe.com/tiobe-index/python/

[15] Python Foundation. (2019) The Very High Level Layer — Python 3.7.2 documentation. Retrieved January 10, 2019, from https://docs.python.org/3/c-api/veryhigh.html

[16] Xtrap Game Protection. (2019). Retrieved January 10, 2019, from http://www.wiselogic.co.kr/

[17] GameGuard. (2019) nProtect GameGuard. Retrieved January 10, 2019, from http://gameguard.nprotect.com/en/index.html

[18] Microsoft. (2019) x64 calling convention — Microsoft Docs. Retrieved January 11, 2019, from https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2017

[6]Thread Local Storage