Shark Cage Project

Documentation of code and work

Peter Benčík

Francisco Javier García Parrales

Etienne Gramlich

Raquel Romero Sanabria

Index

1. Introduction

In this documentation, we are going to explain the process of design, development and deployment of the desktop application ‘Shark Cage’, a project for the Software Security course in HTWG Konstanz.

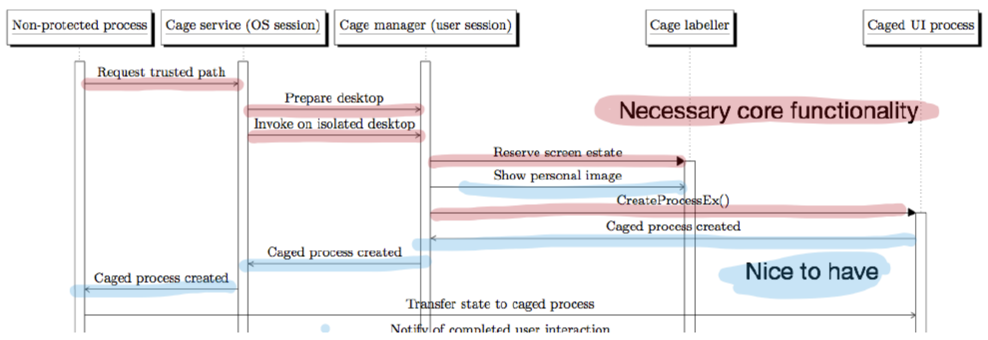
This project contains windows programs to isolate a program in an own desktop to prevent malware (without operative system privileges) to capture user input or sreenshots. It consists of 3 sub-programs: The Cage Service, Cage Manager and user interface. The 3 parts communicate over a TCP connection to interchange messages.

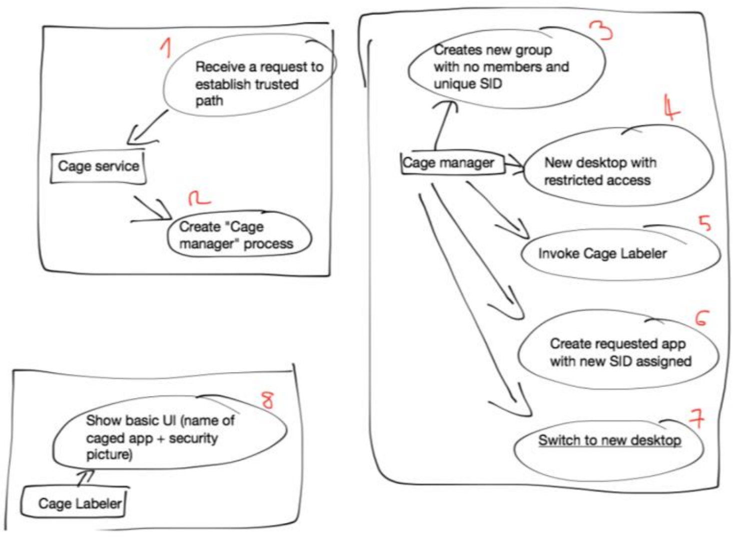
The development group was initially and mainly formed by exchange students and a initial size of 9 students. At the middle-end of the development of this project the group has finally resulted in 4 students. Due to this problem at the end of the development, our initial thoughts and tasks management were firstly made for 9 total developers, which carried us to cancel some tasks and functionalities.

In this document we are going to explain our first thoughts to start developing the project, how it was managed and its results, explaining the parts of the project as could be the code.

2. First thoughts

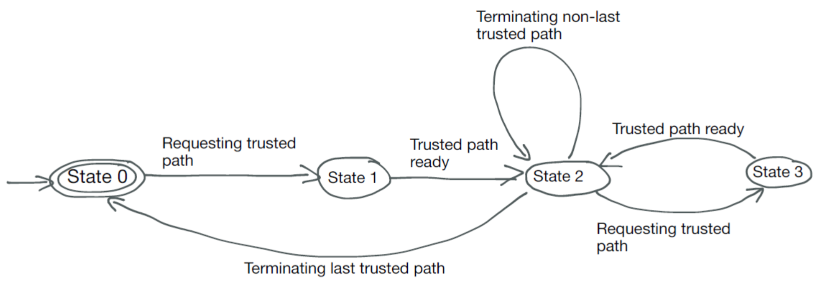
Here we are going to explain how the project was initially thought and how we decided its structure in order to develop it step by step. The project was divided in 3 sub-programs: Cage Service, Cage Manager and Cage labeller / UI. Now, we are going to explain the structure of the poject workflow.




1
Receive a request to establish trusted path
Cage service
2
Create "Cage manager" process
3
Creates new group with no members and unique SID
4
Cage manager
New desktop with restricted access
5
Invoke Cage Labeler
6
Create requested app with new SID assigned
7
Switch to new desktop
8
Show basic UI (name of caged app + security picture)
Cage Labeler

These two pictures presents a high level overview of the system, introducing its structure, how the responsibilities will be delegated to different parts of the system and in which order are they going to be executed. The sequence diagram (first image) represents the original design of the system proposed and the tweaks made to it for it to suit our needs. Use case diagrams on the second image represent individual executables in the project, with the responsibilities of each of them defined as use cases. The numbering represents the sequence in which they shall be executed in runtime.

Shark Cage Service status management diagram



Legend:

State 0: No trusted-path environment is active
State 1: Preparation of first trusted-path environment in progress
State 2: One or more trusted-path environments are active
State 3: Preparation of second or next trusted path environment

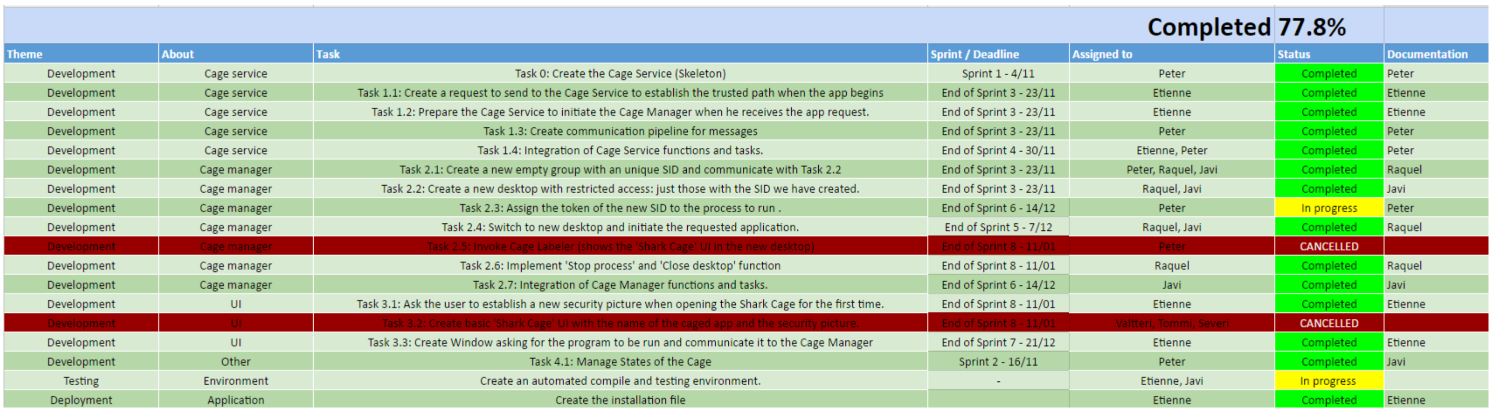
Edges: outgoing edges represent the only messages accepted by Shark Cage Service in that state

All messages received in states 1 and 3 are postponed till the finish and then evaluated in state 2
(Thus restricting the amount of parallelism and possible race-condition problems)

This image represents the service module of the application as a state machine, precisely as a finite deterministic automaton. This diagram was created to clearly define (and restrict) the amount of parallelism in the runtime and provide a guideline for implementing it.

3. Task management

a. Task division



Completed 77.8%

Theme	About	Task	Sprint / Deadline	Assigned to	Status	Documentation
Development	Cage service	Task 0: Create the Cage Service (Skeleton)	Sprint 1 - 4/11	Peter	Completed	Peter
Development	Cage service	Task 1.1: Create a request to send to the Cage Service to establish the trusted path when the app begins	End of Sprint 3 - 23/11	Etienne	Completed	Etienne
Development	Cage service	Task 1.2: Prepare the Cage Service to initiate the Cage Manager when he receives the app request.	End of Sprint 3 - 23/11	Etienne	Completed	Etienne
Development	Cage service	Task 1.3: Create communication pipeline for messages	End of Sprint 3 - 23/11	Peter	Completed	Peter
Development	Cage service	Task 1.4: Integration of Cage Service functions and tasks.	End of Sprint 4 - 30/11	Etienne, Peter	Completed	Peter
Development	Cage manager	Task 2.1: Create a new empty group with an unique SID and communicate with Task 2.2	End of Sprint 3 - 23/11	Peter, Raquel, Javi	Completed	Raquel
Development	Cage manager	Task 2.2: Create a new desktop with restricted access: just those with the SID we have created.	End of Sprint 3 - 23/11	Raquel, Javi	Completed	Javi
Development	Cage manager	Task 2.3: Assign the token of the new SID to the process to run .	End of Sprint 6 - 14/12	Peter	In progress	Peter
Development	Cage manager	Task 2.4: Switch to new desktop and initiate the requested application.	End of Sprint 5 - 7/12	Raquel, Javi	Completed	Raquel
Development	Cage manager	Task 2.5: Invoke Cage Labeler (shows the 'Shark Cage' UI in the new desktop)	End of Sprint 8 - 11/01	Peter	CANCELLED	
Development	Cage manager	Task 2.6: Implement 'Stop process' and 'Close desktop' function	End of Sprint 8 - 11/01	Raquel	Completed	Raquel
Development	Cage manager	Task 2.7: Integration of Cage Manager functions and tasks.	End of Sprint 6 - 14/12	Javi	Completed	Javi
Development	UI	Task 3.1: Ask the user to establish a new security picture when opening the Shark Cage for the first time.	End of Sprint 8 - 11/01	Etienne	Completed	Etienne
Development	UI	Task 3.2: Create basic 'Shark Cage' UI with the name of the caged app and the security picture.	End of Sprint 8 - 11/01	Valtteri, Tommi, Severi	CANCELLED	
Development	UI	Task 3.3: Create Window asking for the program to be run and communicate it to the Cage Manager	End of Sprint 7 - 21/12	Etienne	Completed	Etienne
Development	Other	Task 4.1: Manage States of the Cage	Sprint 2 - 16/11	Peter	Completed	Javi
Testing	Environment	Create an automated compile and testing environment.	-	Etienne, Javi	In progress	
Deployment	Application	Create the installation file		Etienne	Completed	Etienne

b. Workload of tasks by working percentage

Task / Student	Francisco Javier García Parrales	Raquel Romero Sanabria	Peter Benčík	Etienne Gramlich
Task 0	NA	NA	100%	NA
Task 1.1	NA	NA	NA	100%
Task 1.2	NA	NA	NA	100%
Task 1.3	NA	NA	100%	NA
Task 1.4	NA	NA	50%	50%
Task 2.1	33%	33%	33%	NA
Task 2.2	50%	50%	NA	NA
Task 2.3	NA	NA	100%	NA
Task 2.4	50%	50%	NA	NA
Task 2.5	*Cancelled*			
Task 2.6	NA	100%	NA	NA
Task 2.7	100%	NA	NA	NA
Task 3.1	NA	NA	NA	100%
Task 3.2	*Cancelled*			
Task 3.3	NA	NA	NA	100%
Task 4.1	NA	NA	100%	NA

*NA: Not assigned (initially)

4. Code explanation by tasks

Here we will explain how were these tasks developed in our final code project. This consists in 4 sub-projects: CageService, CageManager, StarterCMD and IconSelectDialog. We highly recommend to read this while looking at the code.

Task 1.1: Create a request to send to the Cage Service to establish the trusted path when the app begins

The Cage Manager waits for messages from the user interface by making blocking calls to NetworkManager::listen() in the method ServiceWorkerThread(LPVOID lpParam). The NetworkManager has to be initialized differently in the different parts of the application. The three possibilities are UI, SERVICE and MANAGER for the respective parts. This needs to be done because the TCP connections we used only allow two endpoints. So we had to use different ports for the two connections.

The received messages will be parsed in CageService::handleMessage(). Depending on the received message the service starts or stops the Manager or forwards the message to the Manager.

Task 1.2: Prepare the Cage Service to initiate the Cage Manager when he receives the app request.

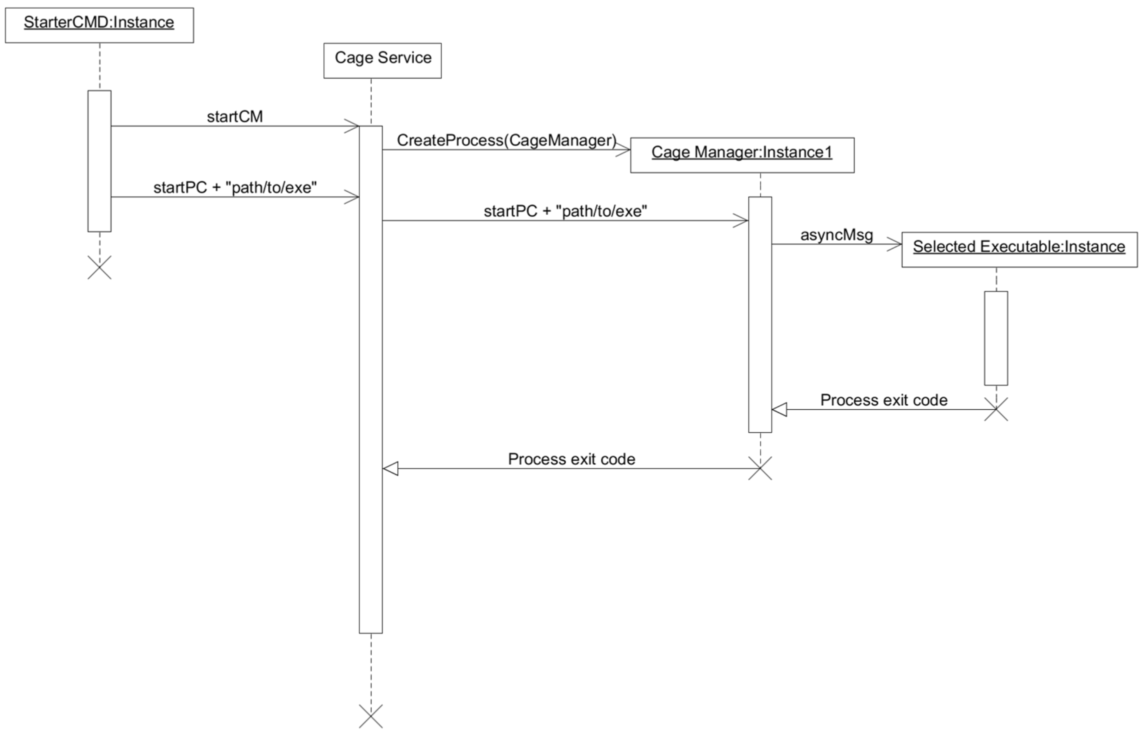
If a message contains the command to start an executable in the Manager, the Service forwards this message to the Manager. The method CageService::handleMessage() forwards the message by itself through the passed Network Manager.

If the Service receives and forwards a message that instructs the Manager to start a program, the Service waits for the Manager to terminate until the next message can be received. This means that only one Cage Manager can run only one program at a time.

Message formats (Both Task 1.1 and 1.2)

To encode different actions in the message there are two files with message formats (MSG_Service.h and MSG_to_manager.h). One is for the messages from the UI to the Cage Service, the other is for messages from the Cage Service to the Cage Manager.

Both contain an enum, an array of strings, an assertion and a function to convert enum values to strings. The enum is there to have some kind of type saftey and have specific names for the commands without remembering the strings to pass over the TCP connection. The array of strings contains the strings to be sent over the TCP connection and must have a representation in the enum. The assertion makes sure that for every enum value a string in the array exists. The function returns the according string for an enum value.



Task 1.3: Create communication pipeline for messages

NetworkManager

This library is responsible for communication between the different executables of application. NetworkManager is build upon non-boost asio library in 1.10.8 version. The library is highly customized for needs of Shark Cage. NetworkManager can be initialized for three different use-cases: UI, SERVICE and MANAGER. When correctly initialized it provides a one-way communication from user interface to service and from service to the manager over a TCP connection. The UI listens for messages on a port 1337, the service on a port 1338 and a manager on a port 1339. The NetworkManger library can be combined freely with any other TCP communication library as long as the communication conventions are being kept.

StatusManager

This file provides a definition for an object holding a current status of a service (form a state machine perspective described in a document “Cage service stages draft.pdf”.

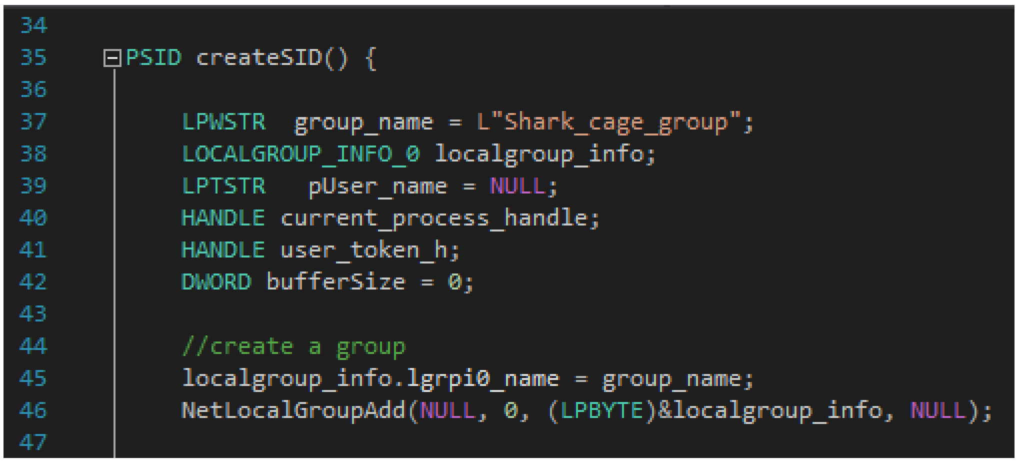
The type *DeviceStatus* enumerates all possible states of the cage service.

The instance of *StatusManager* class represents an object responsible for holding a current status. The only methods provided are getters and setters. Upon construction, the initial status is STATE_0.

Task 2.1: Create a new empty group with an unique SID and communicate with Task 2.2.

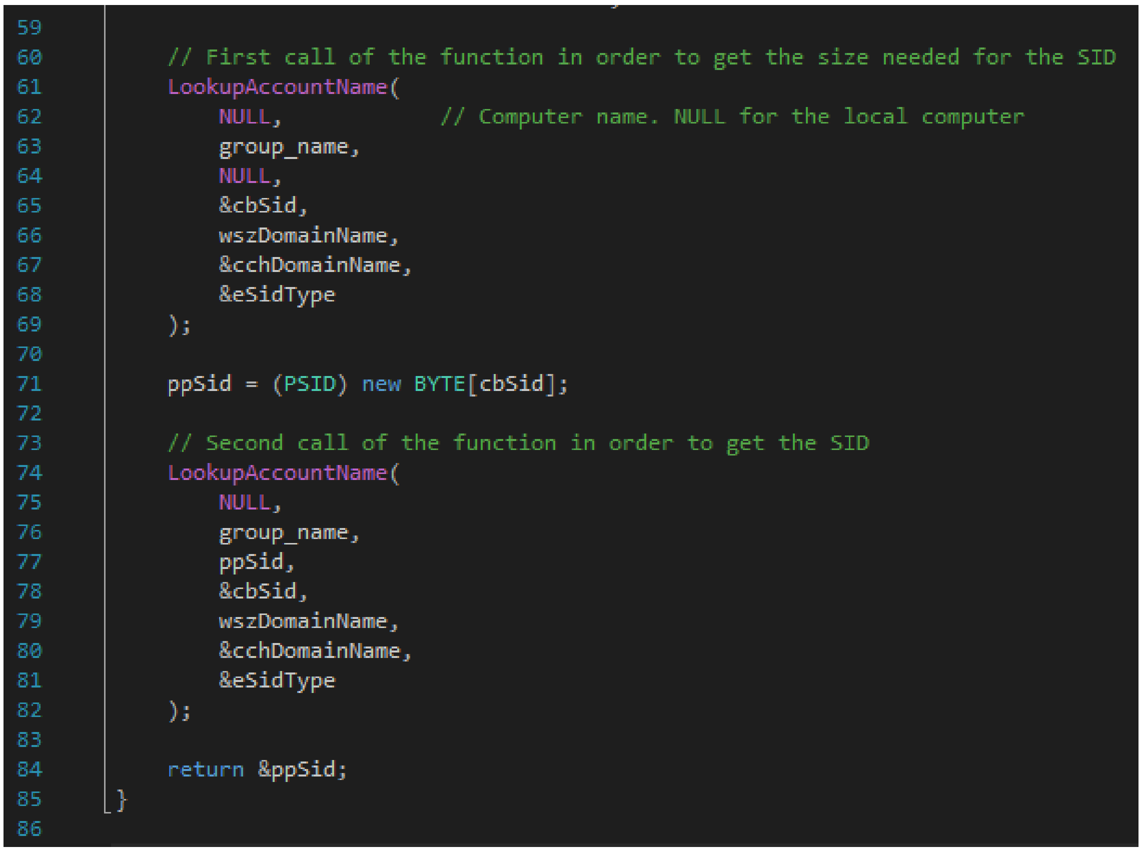
The createSID () function performs this task.

As we can see in the code, the first thing we do is assign the group information to the localgroup_info variable (in our case, basically the group name) and then add the local group, using the NetLocalGroupAdd () function.



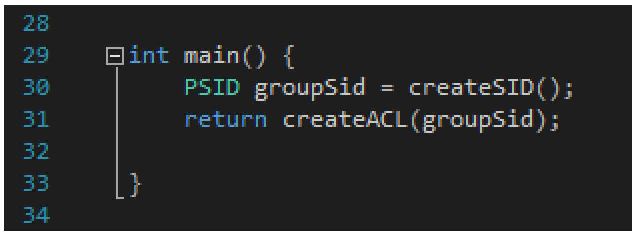
```

PSID createSID() {

    LPWSTR  group_name = L"Shark_cage_group";
    LOCALGROUP_INFO_0 localgroup_info;
    LPTSTR   pUser_name = NULL;
    HANDLE current_process_handle;
    HANDLE user_token_h;
    DWORD bufferSize = 0;

    //create a group
    localgroup_info.lgrpi0_name = group_name;
    NetLocalGroupAdd(NULL, 0, (LPBYTE)&localgroup_info, NULL);

```

After that, we call the LookupAccountName () function twice, passing the name of the created group by parameter. In the first call, we just get the size of the SID that we are going to need. In the second call, passing this recently obtained parameter, we finally get the SID of the previously created group.



```

    // First call of the function in order to get the size needed for the SID
    LookupAccountName(
        NULL,            // Computer name. NULL for the local computer
        group_name,
        NULL,
        &cbSid,
        wszDomainName,
        &cchDomainName,
        &eSidType
    );

    ppSid = (PSID) new BYTE[cbSid];

    // Second call of the function in order to get the SID
    LookupAccountName(
        NULL,
        group_name,
        ppSid,
        &cbSid,
        wszDomainName,
        &cchDomainName,
        &eSidType
    );

    return &ppSid;
}

```

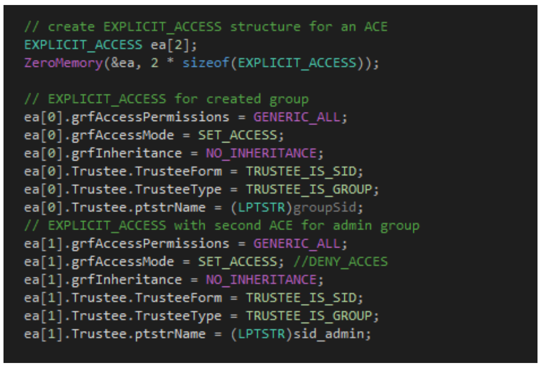
So, finally, we created the group and returned the SID. The point now is to communicate this task with the creation of a restricted desktop. In order to do this, we call createSID() from the main() function, get the SID and call the createACL() function, passing the SID by parameter.



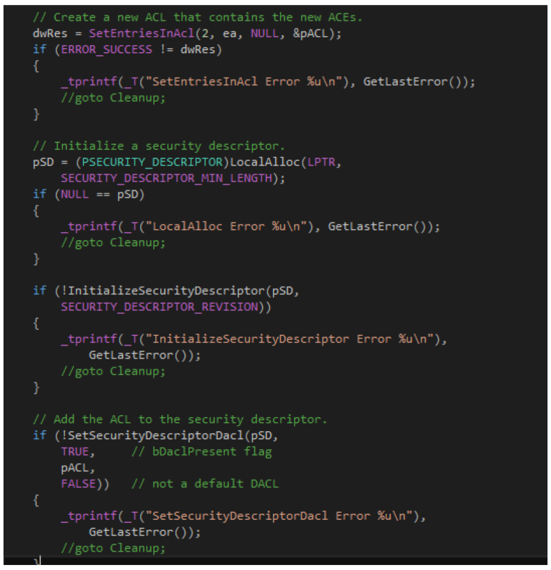
```
int main() {
    PSID groupSid = createSID();
    return createACL(groupSid);

}
```

Task 2.2: Create a new desktop with restricted access: just those with the SID we have created.

The work here consists in creating an ACL and assign it to a new Desktop that we will initiate at the end. In this ACL (Access Control List) we are going to add two entries (ACE), specifying the rights access for the administrator and the new group recently created (by the SID we can get from task 2.2).

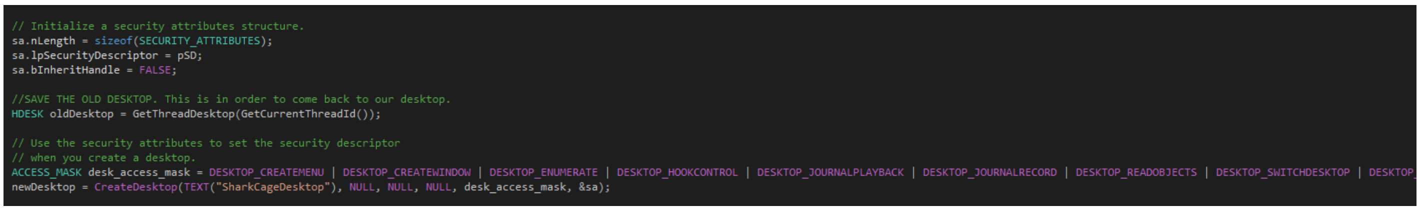


```
// create EXPLICIT_ACCESS structure for an ACE
EXPLICIT_ACCESS ea[2];
ZeroMemory(&ea, 2 * sizeof(EXPLICIT_ACCESS));

// EXPLICIT_ACCESS for created group
ea[0].grfAccessPermissions = GENERIC_ALL;
ea[0].grfAccessMode = SET_ACCESS;
ea[0].grfInheritance = NO_INHERITANCE;
ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[0].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
ea[0].Trustee.ptstrName = (LPTSTR)groupSid;
// EXPLICIT_ACCESS with second ACE for admin group
ea[1].grfAccessPermissions = GENERIC_ALL;
ea[1].grfAccessMode = SET_ACCESS; //DENY_ACCES
ea[1].grfInheritance = NO_INHERITANCE;
ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[1].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
ea[1].Trustee.ptstrName = (LPTSTR)sid_admin;
```



```
// Create a new ACL that contains the new ACEs.
dwRes = SetEntriesInAcl(2, ea, NULL, &pACL);
if (ERROR_SUCCESS != dwRes)
{
    _tprintf(_T("SetEntriesInAcl Error %u\n"), GetLastError());
    //goto Cleanup;
}

// Initialize a security descriptor.
pSD = (PSECURITY_DESCRIPTOR)LocalAlloc(LPTR,
    SECURITY_DESCRIPTOR_MIN_LENGTH);
if (NULL == pSD)
{
    _tprintf(_T("LocalAlloc Error %u\n"), GetLastError());
    //goto Cleanup;
}

if (!InitializeSecurityDescriptor(pSD,
    SECURITY_DESCRIPTOR_REVISION))
{
    _tprintf(_T("InitializeSecurityDescriptor Error %u\n"),
        GetLastError());
    //goto Cleanup;
}

// Add the ACL to the security descriptor.
if (!SetSecurityDescriptorDacl(pSD,
    TRUE,     // bDaclPresent flag
    pACL,
    FALSE))   // not a default DACL
{
    _tprintf(_T("SetSecurityDescriptorDacl Error %u\n"),
        GetLastError());
    //goto Cleanup;
}
```

After we create the Explicit Access entries (ACEs), we add them to the ACL and assign that ACL to our new Desktop.



```
// Initialize a security attributes structure.
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = pSD;
sa.bInheritHandle = FALSE;

//SAVE THE OLD DESKTOP. This is in order to come back to our desktop.
HDESK oldDesktop = GetThreadDesktop(GetCurrentThreadId());

// Use the security attributes to set the security descriptor
// when you create a desktop.
ACCESS_MASK desk_access_mask = DESKTOP_CREATEMENU | DESKTOP_CREATEWINDOW | DESKTOP_ENUMERATE | DESKTOP_HOOKCONTROL | DESKTOP_JOURNALPLAYBACK | DESKTOP_JOURNALRECORD | DESKTOP_READOBJECTS | DESKTOP_SWITCHDESKTOP | DESKTOP_
newDesktop = CreateDesktop(TEXT("SharkCageDesktop"), NULL, NULL, NULL, desk_access_mask, &sa);
```

Task 2.3: Assign the token of the new SID to the process to run.

(THIS IS A WORK IN PROGRESS - NOT OPERATIONAL YET!)

TokenManipulation

This library was created in order to encapsulate the token manipulation into a singular unit. The goal of the library is to create a token, that is the only token with the access rights to the secure desktop. There are two functions accomplishing the same task in this library. (due the "In progress" state)

The first one (getModifiedToken) takes approach by obtaining an existing token and exploiting it and manipulating the internal structures by artificially adding a dummy group into the token and then swapping the dummy_group with the special access group. Fixing the integrity issues in a token is still under development.

The second function (constructUserTokenWithGroup) uses the NT-api functions in cooperation with a driver development tools to specify and construct the token "to measure" - with all the information in the token adjusted to needs of the application. Specifying the group parameters, along with the usage of undocumented NT-api renders this function not operational yet.

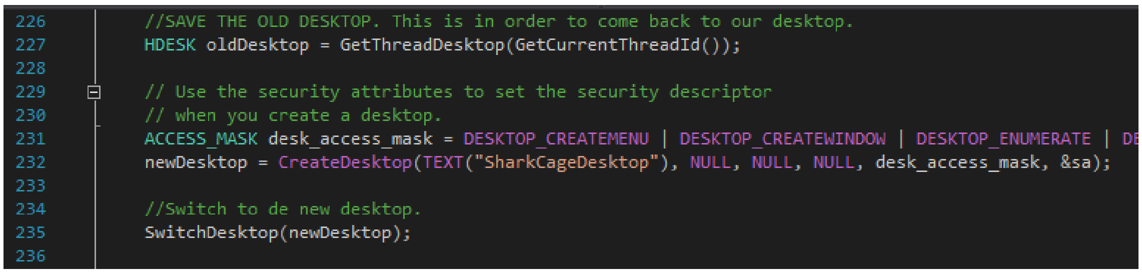
LOCAL_SYSTEM account privileges are required for this library to execute correctly.

Task 2.4: Switch to new desktop and initiate the requested application.

Within the createACL () function, after assigning the token to the SID of the created group and after including that entry in a descriptor, we create a new desktop with that assigned descriptor.

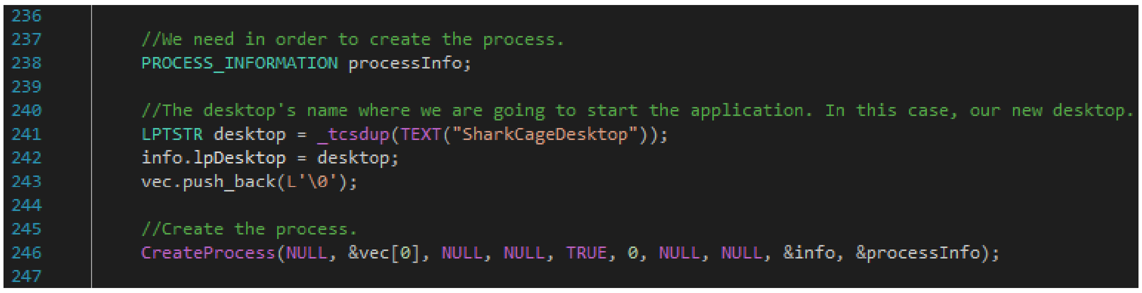
We first obtain a handle from the current desktop, in order to be able to come back later. Next, we create the desktop using the CreateDesktop() function, passing as parameters an access mask, the name of the desktop and the descriptor in which we have previously assigned the permissions.

After that, we switch from the current desktop to the desktop created using the SwitchDesktop() function.



```
    //SAVE THE OLD DESKTOP. This is in order to come back to our desktop.
    HDESK oldDesktop = GetThreadDesktop(GetCurrentThreadId());

    // Use the security attributes to set the security descriptor
    // when you create a desktop.
    ACCESS_MASK desk_access_mask = DESKTOP_CREATEMENU | DESKTOP_CREATEWINDOW | DESKTOP_ENUMERATE | DE
    newDesktop = CreateDesktop(TEXT("SharkCageDesktop"), NULL, NULL, NULL, desk_access_mask, &sa);

    //Switch to de new desktop.
    SwitchDesktop(newDesktop);
```

After that, we must create a process on the new desktop. In order to do this, we have to use the CreateProcess() function, but we have to pass as a parameter certain information regarding the new process, in which we specify that we want it to run on the recently created desktop.



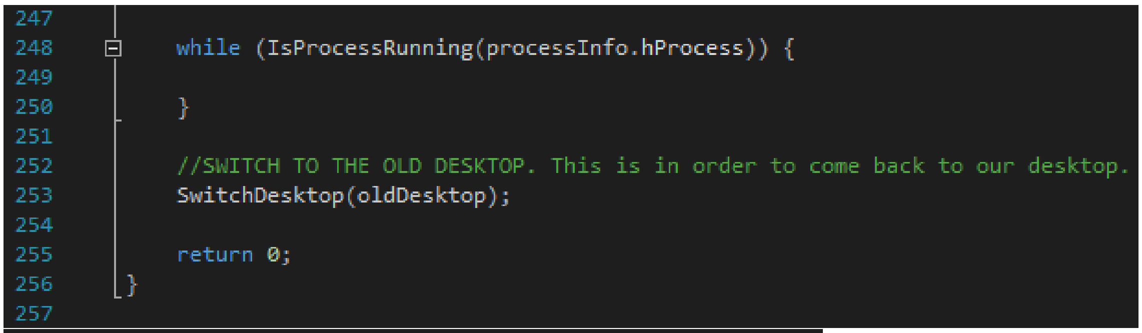
```
    //We need in order to create the process.
    PROCESS_INFORMATION processInfo;

    //The desktop's name where we are going to start the application. In this case, our new desktop.
    LPTSTR desktop = _tcsdup(TEXT("SharkCageDesktop"));
    info.lpDesktop = desktop;
    vec.push_back(L'\0');

    //Create the process.
    CreateProcess(NULL, &vec[0], NULL, NULL, TRUE, 0, NULL, NULL, &info, &processInfo);
```

We must also pass the path of the application we want to run, which we receive by message, through a function implemented in another part of the code.

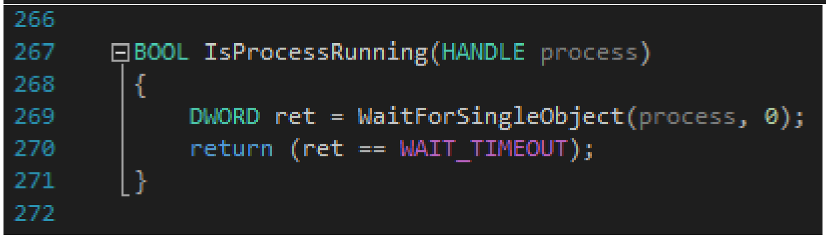
Task 2.6: Implement 'Stop process' and 'Close desktop' function.

The next thing we have to do is to control that we can go back to the old desktop as soon as the user closes the process.

In order to do that, we use a blocking instruction. A loop keeps running while the process is active. To do this, we have created a function called isProcessRunning(), which returns a boolean. When the user closes the process, we exit the loop and return to the old desktop, using the SwitchDesktop() function again.



```
    while (IsProcessRunning(processInfo.hProcess)) {

    }

    //SWITCH TO THE OLD DESKTOP. This is in order to come back to our desktop.
    SwitchDesktop(oldDesktop);

    return 0;
}
```



```
BOOL IsProcessRunning(HANDLE process)
{
    DWORD ret = WaitForSingleObject(process, 0);
    return (ret == WAIT_TIMEOUT);
}
```

Task 2.7: Integration of Cage Manager functions and tasks.

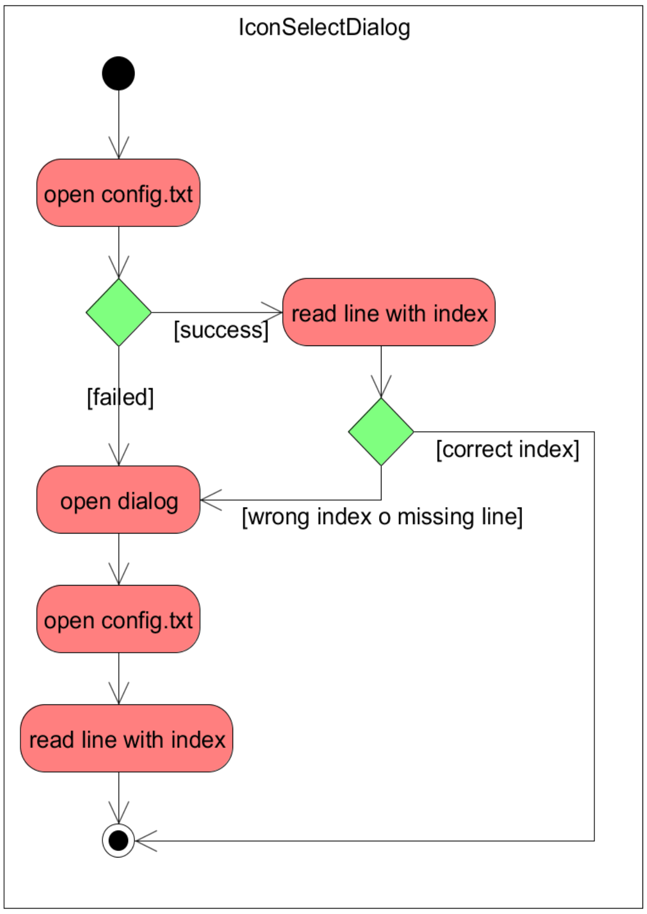
In the CageManager sub-project were added all the functionalities of it like Creating the ACL entries, waiting the message of the application path to execute and creating and switching the Desktop, in a main function that makes calls to the other ones.

Task 3.1: Ask the user to establish a new security picture when opening the Shark Cage for the first time.

The project *imageSelectDialog* contains a custom user interface with three picture boxes and respective click handlers that write the index of the according to the configuration file. With the current three images the indices are numbers from 0 to 2.

Currently the images are compiled to the dialog and do not have to be in the folder with the binaries, because they are neither be accessed by the Service nor the Manager.

The next step is to pass the index of the image to the Manager and display the according image on the newly created desktop.



Task 3.3: Create Window asking for the program to be run and communicate it to the Cage Manager

The program for this UI needs an instance of NetworkManager to start the Cage Manager and to send the path to the executable to be started. In this case the NetworkManager is initialized with UI.

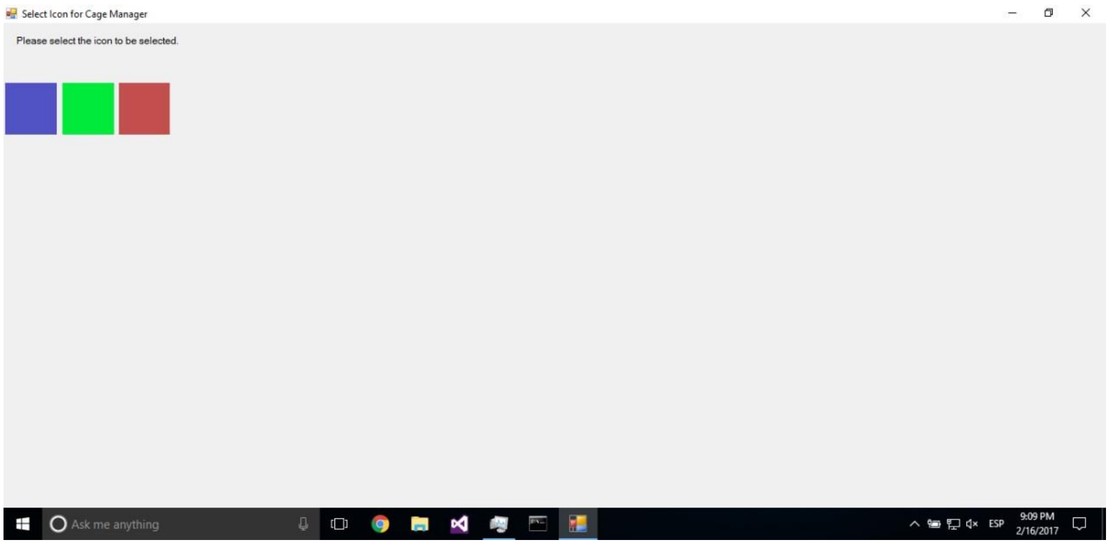
It opens two dialogs: the first asks if the Manager has to be started at all and the second dialog is a file selector dialog that displays only .exe files.

If the first dialog is answered with yes, the Manager is started, and after selecting the executable file the path will be sent to the Cage Service, which forwards the message to the Manager.

5. Results

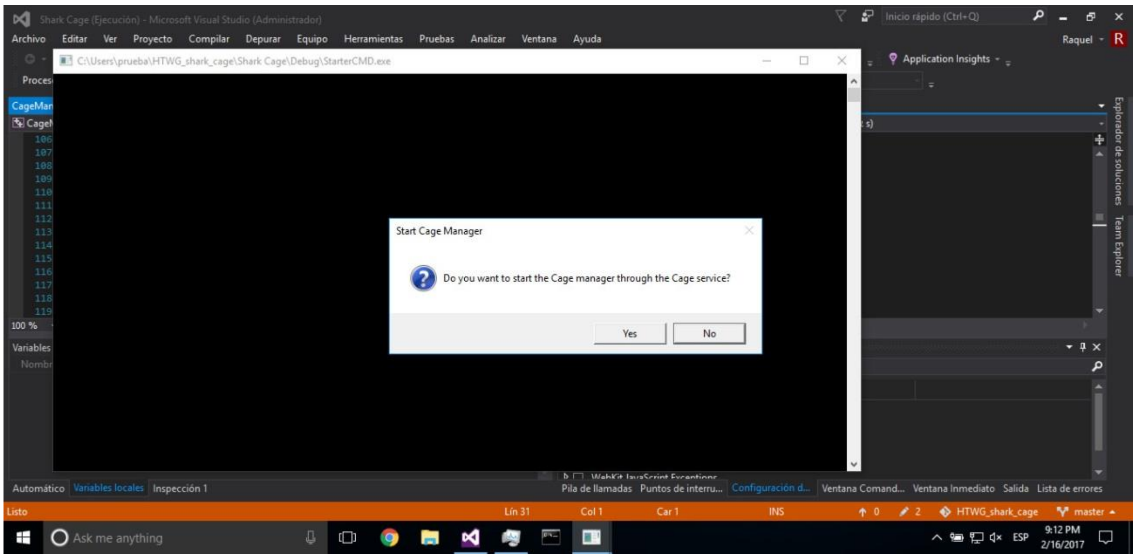
Here we are going to show the results of the project, followed by screenshots of a demonstration. As we could see in the tasks, some functionalities were not added because of lack of time and members leaving, but we could finish the core functionalities and make it into an executable application:

It first asks the user to establish a security icon (that would be displayed in the Shark Cage UI), only if it is the first time he/she executes this.



Please select the icon to be selected.

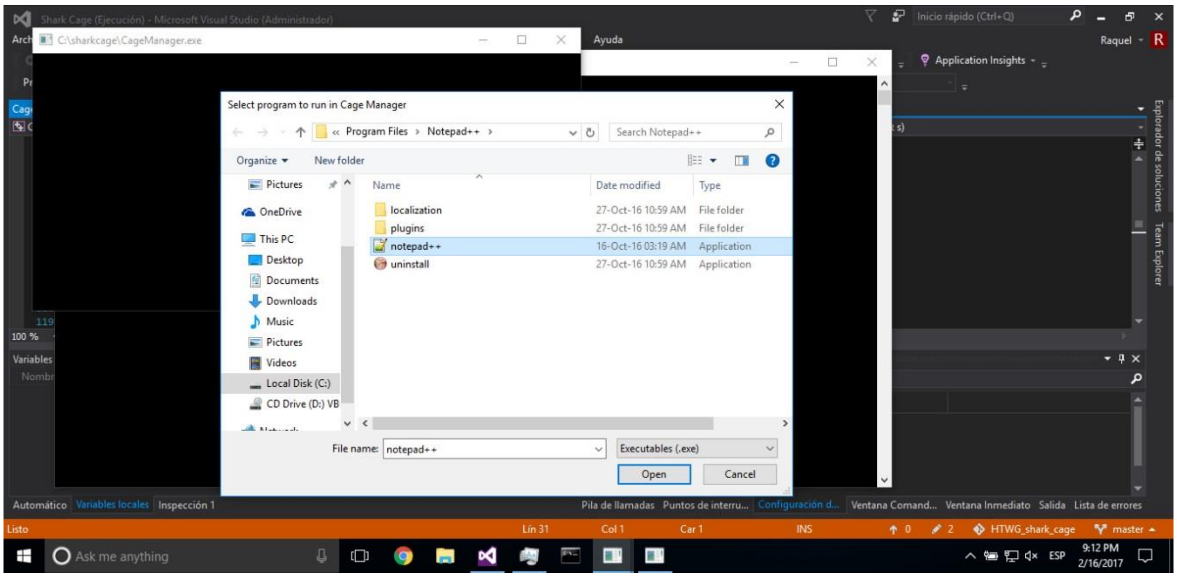
Secondly, the selected icon is saved in order to use it later, and a dialog box is prompted in order to continue. We must choose Yes in order to continue.



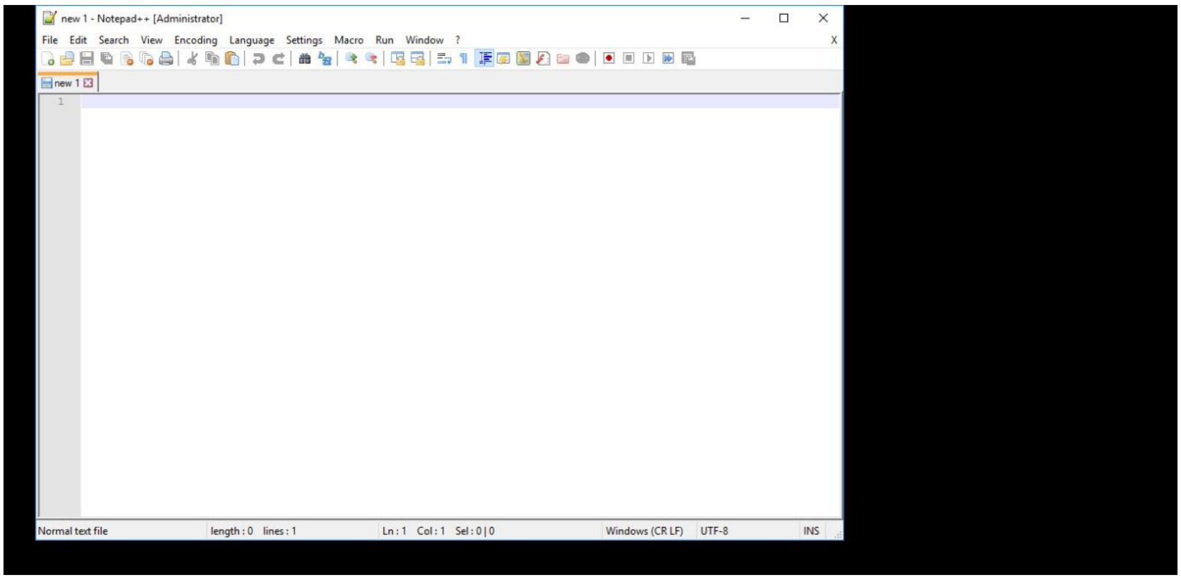
C:\Users\prueba\HTWG_shark_cage\Shark Cage\Debug\StarterCMD.exe

Do you want to start the Cage manager through the Cage service?

After that, the program selector will be shown, so the user can choose the program he wants to run in the Shark Cage (only .exe are eligible).



Finally, the program is successfully executed and shown in our new Shark Cage Safe-Desktop, and it will be close when the user chooses to close the open program.


new 1 - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Macro Run Window ?
new 1
Normal text file
length : 0 lines : 1
Ln : 1 Col : 1 Sel : 0 | 0
Windows (CR LF) UTF-8 INS