The A.R.F Project

Anti-Reversing Framework v1.1

Author: Economou Kyriakos

Contact: arfproject@hotmail.com

Programming Language: C++

Compatibility: Win XP SP1 or above

IDE plug 'n compile: MS Visual Studio (tested with 2008/2010 editions), Embarcadero C++ builder (tested with 2010 edition).

Release Date: 17d/01m/2011

Intro

After having dedicated some years in breaking software protections I have arrived to some conclusions regarding how things work among software developing companies regarding this matter.
A company that produces any type of software has always in mind the quality of the final product to be released. This means that its coders dedicate all of their time in developing the software that is going to be released by searching for bugs and errors and by improving the code of the software.
However, when all this has come into an end, and it is time to release the software, all companies have to, or at least they should, deal with another big decision which is of course the "how" their product it is going to be protected by crackers.
For the big companies that have more money than time in their hands it is more easy to decide because their budget can afford any type of commercial protections and it is only a matter of taste regarding the quality of the protection that it is going to be chosen.
However, even if these companies are able to choose among a variety of protections, the impact of this decision can be destructive in zero time. Commercial protectors offer much more quality than the free of the kind but they have a major flaw.
This flaw relies on the fact that before a company applies a well known commercial protector, others have already done it which means that probably it has been already reversed. So why should you spend money for a protection that you know that it has been already cracked?!?

From the other hand, crackers have usually much more time than money in their hands and this is what makes them dangerous. They will usually try to break the protection by themselves or read a tutorial about it and use it as a guide. In any case it is just a matter of time before they say “Game Over!”
So then, why companies do not create their own custom protections?
The answer is simple, it requires time, money and people who know the basics of reverse engineering, in other words how a cracker would attack the protection of the software. All this would normally require a separate department of employees dedicated to the development of the protection itself, but companies don’t seem to approve this idea and you know the results.
But crackers, won’t they crack that protection too?
The answer is “Yes!” but as we have already said the entire game goes around a more specific and relative concept which is the “time”. A custom protection must be analyzed each time from scratch and if it is well developed it could take to crackers much more time than you may think, so the company will have the opportunity to gain time and money before the game arrives into an end.
So what if a company wants to use a commercial protector?
Even if they have the flaw that we discussed before, many commercial protectors are very good and can keep a big amount of crackers especially newbies away. But even in this case, why you should rely just to the protector itself?!?
Well, you shouldn’t and here it is where the **A.R.F Project** comes into the game. You could use it in order to add an extra layer of protection inside the code of your application itself that will protect your software when the protector will not. Furthermore, it can be a good solution for single developers and companies that may not afford neither in terms of money the use of a commercial protector nor in terms of time the development of a custom protection from scratch.
The A.R.F Project is Free and comes with full source code and documentation of the available anti-reversing methods.
This gives also the possibility to the developers to modify these methods at will, combine them together and even more get inspired to create something better in less time than ever before.
In addition, through the **A.R.F Project** you can understand how some of the most famous anti-reversing tricks work and learn how an attacker would attempt to bypass them, which will help you create your own custom software protection.
The available methods will constantly be updated and more methods are going to be added in the days to come.

The A.R.F project offers plug ‘n compile compatibility with MS Visual Studio (tested with 2008/2010 edition) and Embarcadero C++ Builder (tested with 2010 edition).

Greetings: This project is dedicated to all the people that I really respect for their devotion to their passions without thinking about the consequences.

A big salute to my friends Yiannis, Alexandros, Panos, Kyprianos and Anna.

Enjoy,
Economou Kyriakos

Available Classes & Methods

The layout is:

- **Class**

i) method

- **DirectDebuggerDetection**

i) `bool DebuggerPresent()`

ii) `int RemoteDebuggerPresent()`

- **IndirectDebuggerDetection**

i) `bool DebugString()`

ii) `int OpenServicesProcess()`

- **WindowDebuggerDetection**

i)
```
bool SpecificWindowNameDetection(string
                windowname)
```
ii) `bool SpecificWindowClassDetection(string classname)`

iii) `void SetListSize()`

iv) `int GetListSize()`

v) `bool ListWindowClassDetection(string * arraymemlocation , int listsize)`

- **ProcessDebuggerDetection**

i) `string * SetProcessList()`

ii) `void SetListSize()`

iii) `int GetListSize()`

iv) `int ProcessDetection(string * arraymemlocation , int listsize)`

- **ModuleDebuggerDetection**

i) `string * SetModulesList()`

ii) `void SetListSize()`

iii) `int GetListSize()`

iv) `int ModuleDetection(string * arraymemlocation, int listsize)`

- **ParentProcessDetection**

i) `int CheckParentProcess()`

- **CodeTraceTimeDetection**

i) `DWORD StartExecutionTime()`

ii) `DWORD EndExecutionTime()`

iii) `DWORD GetTimeLimit()`

iv) `DWORD GetTotalTime()`

v) `void SetStartTime()`

vi) `void SetEndTime()`

vii) `void SetTimeLimit()`

viii) `void SetTotalTime()`

ix) `bool IsCodeBeingTaced()`

§ **HardwareBreakPointDetection**

i) `int HwdBreakPoint()`

§ **ApiBreakPointDetection**

i) `int ApiBreakPoint(char * DLL, char * API)`

§ **SehDbgDetection**

i) `bool` CloseHandleExcepDetection(HANDLE invalid)

ii) `bool` SingleStepExcepDetection()

§ **AntiAttach**

i) `int AntiAttachSet()`

ii) `void AntiAttachSelfDebug()`

Methods Documentation

Class: DirectDebuggerDetection

1. `DirectDebuggerDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `bool DebuggerPresent()`

This method uses the win API **IsDebuggerPresent** in order to directly detect if the process is being debugged.

This method returns **true** if a debugger has been detected, otherwise it returns **false**.

3. `int RemoteDebuggerPresent()`

This method uses the win API **CheckRemoteDebuggerPresent** in order to directly detect if the process is being debugged.

This method returns **1** if a debugger has been detected, **0** if a debugger has not been detected, **-1** if an error occurs while trying to obtain a valid handle to our process with the necessary access rights, and **-2** if the win API fails.

*****Note***: Both methods will trigger a "false alarm" while you debug the code in your IDE.**

Class: IndirectDebuggerDetection

1. `IndirectDebuggerDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `bool DebugString()`

This method uses the win API **OutputDebugString** in order to indirectly detect if the process is being debugged. If a debugger is not attached then the return value will be 0 or 1 depending on the windows version. By retrieving the return value from EAX immediately after the execution goes back to our code we can determine if the process is being debugged.

This method returns **true** if a debugger has been detected, otherwise returns **false**.

*****Note***: This method will trigger a "false alarm" while you debug the code in your IDE.**

3. `int OpenServicesProcess()`

Our process normally does not have the **SeDebugPrivilege** enabled, which means that we are not able to obtain a valid handle through **OpenProcess** win API to a vital system process such as **"services.exe"** with **process_all_access rights**.

In case a debugger is attached to our process ***and*** has enabled the **SeDebugPrivilege** we will manage to obtain such a handle so we can presume that our process is being debugged.

This method returns **1** if a debugger has been detected, **0** if a debugger has not been detected, **-1** if we failed to obtain a snapshot of all running processes, and **-2** if we failed to retrieve info regarding the first running process.

*****Note***: This detection method should not be used in case that you have programmatically enabled the SeDebugPrivilege for the needs of your application.**
The debugger of your IDE may also enable the SeDebugPrivilege to the process, which means that in that case this method will trigger a "false alarm" while you debug your code.

Class: WindowDebuggerDetection

1. `WindowDebuggerDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `bool SpecificWindowNameDetection(string windowname)`

This method can be used in order to detect the window of a debugger or a specific reversing tool through its title name. This it is achieved by trying to obtain a valid handle to its top-level window by using the win API **FindWindow**.

3. `bool SpecificWindowClassDetection(string classname)`

This method can be used in order to detect the window of a debugger or a specific reversing tool through its class name. This it is achieved by trying to obtain a valid handle to its top-level window by using the win API **FindWindow**.

Both methods will return **true** if the desired window has been detected, otherwise they will return **false**.

4. `string * SetReverseToolsList()`

This method will fill a dynamic array with a predefined list of some popular window class names of debuggers and reversing tools and it will return a pointer to it.

It is the first parameter of `ListWindowClassDetection` method.

You may modify the list, but in case you do so, you must set the correct size in the `SetListSize` method.

5. `void SetListSize()`

This method sets the size of the list and it is used by `GetListSize()` method.

You should modify the size only in case that you have modified the size of the list inside the `SetReverseToolsList()` method itself.
For example, in case that you have deleted or added one or more class names of windows that you want to be detected through the `ListWindowClassDetection method.`

6. `int GetListSize()`

This method will use the `SetListSize` method and it will return the size of the list.

It is used as the second parameter of `ListWindowClassDetection` method.

7. `bool ListWindowClassDetection(string * arraymemlocation , int listsize)`

This method is used in order to check through a list of popular windows class names of debuggers and reversing tools if any of them is running.

Check methods above regarding the predefined list.

This method will return **true** if a debugger or a reversing tool from the predefined list gets detected by its top-level window class name, otherwise it will return **false**.

Class: ProcessDebuggerDetection

1. `ProcessDebuggerDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `string * SetProcessList()`

This method will fill a dynamic array with a predefined list of some popular process names of debuggers and reversing tools and it will return a pointer to it.

It is the first parameter of `ProcessDetection` method.

You may modify the list, but in case you do so, you must set the correct size in the `SetListSize` method.

3. `void SetListSize()`

This method sets the size of the list and it is used by `GetListSize()` method.

You should modify the size only in case that you have modified the size of the list inside the `SetProcessList()`method itself.
For example, in case that you have deleted or added one or more process names of that you want to be detected through the `ProcessDetection` method.

4. `int GetListSize()`

This method will use the `SetListSize` method and it will return the size of the list.

It is used as the second parameter of `ProcessDetection` method.

5. `int ProcessDetection(string * arraymemlocation , int listsize)`

This method is used in order to check through a list of popular process names of debuggers and reversing tools if any of them is running.

Check methods above regarding the predefined list.

This method will return **1** if a debugger or a reversing tool from the predefined list gets detected through it process name, **0** if no suspect process has been detected, **-1** if an error occurs while trying to obtain a snapshot of all running processes, and **-2** if an error occurs while trying to retrieve info about the first running process.

Class: ModuleDebuggerDetection

1) `ModuleDebuggerDetection()`

This method is the constructor used in order to create a new instance of the class.

2) `string * SetModulesList()`

This method will fill a dynamic array with a predefined list of some specific modules/plugins of debuggers and reversing tools and it will return a pointer to it.

It is the first parameter of `ModuleDetection` method.

You may modify the list, but in case you do so, you must set the correct size in the `SetListSize` method.

3) `void SetListSize()`

This method sets the size of the list and it is used by `GetListSize()` method.

You should modify the size only in case that you have modified the size of the list inside the `SetModulesList()`method itself.
For example, in case that you have deleted or added one or more modules/plugins names that you want to be detected through the `ModuleDetection` method.

4) `int GetListSize()`

This method will use the `SetListSize` method and it will return the size of the list.

It is used as the second parameter of `ModuleDetection` method.

5) `int ModuleDetection(string * arraymemlocation, int listsize)`

This method will take a list of all running processes and then for each process will go through its loaded modules in order to detect a debugger or a reversing tool through its own loaded modules and/or plugins.

Check methods above regarding the predefined list.

This method will return **1** if a debugger or a reversing tool gets detected through a module name, **0** if no suspect module has been detected, **-1** if an error occurs while trying to obtain a snapshot of all running processes, and **-2** if an error occurs while trying to retrieve info about the first running process.

Class: ParentProcessDetection

1. `ParentProcessDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `int CheckParentProcess()`

This method will retrieve the parent process id of our process and then will verify that this id corresponds to the process id of windows explorer. If it is not then it means that probably our process is being debugged or it has been launched through a loader that will attempt to modify our code on run-time in memory.

This method will return **1** in case the parent of our process is not windows explorer, **0** if the parent is windows explorer, **-1** if an error occurs while trying to obtain a snapshot of all running processes, and **-2** if an error occurs while trying to retrieve info about the first running process.

*****Note***: This method should not be used in case you plan to design an application that it will be launched from another one that you already know that it is legitimate.**
This method will trigger a "false alarm" while you debug the code in your IDE.

Class: CodeTraceTimeDetection

1. `CodeTraceTimeDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `DWORD StartExecutionTime()`

This method is going to use the **GetTickCount** win API in order to retrieve the number of milliseconds that have elapsed since the system was started and the result is going to be stored for later.
It is used by the `SetStartTime` method in order to store the initial result.

This method returns the number of milliseconds that have elapsed since the system was started.

3. `DWORD EndExecutionTime()`

This method is going to use the **GetTickCount** win API in order to retrieve the number of milliseconds that have elapsed since the system was started and the result is going to be stored for later.
It is used by the `SetEndTime` method in order to store the initial result.

This method returns the number of milliseconds that have elapsed since the system was started.

4. `void SetStartTime()`

This method uses the `StartExecutionTime` method and stores the value returned by it.

5. `void SetEndTime()`

This method uses the `EndExecutionTime` method and stores the value returned by it.

6. `void SetTimeLimit()`

This method sets the time limit for execution of the code block that we want to 1000 ms.
You may change it to lower or to a higher value depending on your needs.

This method is used from the `GetTimeLimit` method.

7. `DWORD GetTimeLimit()`

This method returns the value of the time limit by using the `SetTimeLimit` method.

This method is used by the `IsCodeBeingTaced` method.

8. `void SetTotalTime()`

This method sets the total time that has elapsed during the execution of a specified code block of our choice.
It is used by the `GetTotalTime` method.

9. `DWORD GetTotalTime()`

This method returns the value of the total time that has elapsed during the execution of a specified code block of our choice by using the `GetTotalTime` method.

This method is used by the `IsCodeBeingTaced` method.

10. `bool IsCodeBeingTaced()`

This method will use the `GetTotalTime` method in order to retrieve the total time that has elapsed during the execution of a specified code block of our choice and then it will retrieve the time limit that we have set by using the `GetTimeLimit` method.

This method returns **true** if the value returned by `GetTotalTime` method is greater than the value returned by `GetTimeLimit` method which means that our code is being traced, otherwise it returns **false**.

*****Note***: This method may trigger a "false alarm" while you debug the code in your IDE.**

Class: HardwareBreakPointDetection

1. `HardwareBreakPointDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `int HwdBreakPoint()`

This method will use the win API **OpenThread** in order to obtain a valid handle with the necessary access rights to the current thread the functions runs and then it will use the **GetThreadContext** win API in order to check if any of the debug registers **Dr0**, **Dr1**, **Dr2**, **Dr3** is not equal to zero which means that one or more **hardware breakpoints** have been set.

This method returns **1** if a hardware breakpoint is detected, **0** if no hardware breakpoints are detected, **-1** if an error has occurred while trying to obtain a valid handle to the thread, and **-2** if the **GetThreadContext** win API fails.

Class: ApiBreakPointDetection

1. `ApiBreakPointDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `int ApiBreakPoint(char * DLL, char * API)`

This method is used in order to detect any **software breakpoints** at the entry point of an API.
It uses the **LoadLibrary** and **GetProcAddress** win APIs in order to obtain a handle to the dll of our choice in order to retrieve the virtual address of the specified API in memory.

It will then use a **BYTE *** pointer in order to check if in the first 5 bytes of the entry point of the API there is a **0xCC** byte which means that a software breakpoint has been set there.

This method returns **1** if a software breakpoint has been detected, **0** if a software breakpoint has not been detected, **-1** if an error occurs while we try to obtain a valid handle to the dll of our choice, and **-2** if an error occurs while we try to retrieve the virtual address of the selected API.

Class: SehDbgDetection

1. `SehDbgDetection()`

This method is the constructor used in order to create a new instance of the class.

2. `bool CloseHandleExcepDetection(HANDLE invalid)`

In this method we are using **CloseHandle** win API by pushing and invalid handle to it.
If there is no debugger attached to our process **CloseHandle** will just return an error code and the execution will continue.
However, if a debugger is attached then an exception will be raised and execution will be transferred inside our SEH.

3. `bool SingleStepExcepDetection()`

In this method we trigger a single step exception by pushing the EFLAGS to the stack and then setting the trap flag bit.
Finally, we restore them back with the trap flag bit set.
If a debugger is attached it will intercept the exception raised during the execution of the next instruction, so execution will never reach our SEH.

*****Note***: These methods may work or not depending on the configuration of the debugger regarding exceptions.**
For example, regarding the INVALID HANDLE exception in Olly Debugger, in case you don't pass the generated exception to the program, execution will never reach our SEH, so the method will return false.

However, if you have instructed the debugger to automatically ignore this specific exception or you manually pass the exception to the program, then the execution will be transferred to our SEH and the method will return true.
Of course this is just for detection purposes, but think what it can happen in this case if the code inside your SEH creates some havoc to mislead the attacker.

Class: AntiAttach

1. `AntiAttach()`

This method is the constructor used in order to create a new instance of the class.

2. `int AntiAttachSet()`

This method can be used in order to forbid to a debugger to attach to our running process.
More specifically, since it is known that each time a user mode debugger attaches to our process, there will be a call to the APIs **DebugUiRemoreBreakin and DbgBreakPoint (exported by the ntdll)** from our process, we can alter these 2 functions by changing their entry point with a **RET** instruction in order to forbid to the debugger to successfully attach to our process.

This method uses the **OpenProcess** win API in order to obtain a valid handle with read/write access rights to our process and the **LoadLibrary** and **GetProcAddress** in order to retrieve the virtual addresses of the 2 APIs mentioned before.
It will then use the **WriteProcessMemory** win API in order to write a **RET** instruction **(0xC3)** at the beginning of the 2 APIs mentioned before so that they will not be able to be used and the debugger will not manage to attach to our process.

This method returns **1** if the antiattach has been se successfully, **-1** if an error occurs while trying to obtain a handle to our process, **-2** if an error occurs while trying to obtain a handle to the ntdll, **-3** if an error occurs while we try to retrieve the address of the 2 win APIs we are interested to, and **-4** if an error occurs while we try to write a **RET** instruction to the entrypoint of both APIs.

3. `void AntiAttachSelfDebug()`

This method will create a child process and it will debug it on run time.

The execution of the code will take place on the child process in which we will not be able to attach a user mode debugger because it will be already being debugged by its parent process.
This is a more sophisticated anti-attach method and anti-debug method since the normal execution of the does not take place in the process we are currently debugging, but from the child process.

*****Note***: In case you decide to use this anti-attach method keep in mind that you should use it inside your main() function as the first thing to do, or after calling the previous method discussed in order to protect also the parent from attaching a debugger to it on run-time.**
This method should not be used along with the following debugger detection methods because since the child process from which the application will run it will be debugged by the parent process, those methods will trigger a "false alarm":

i) `bool DebuggerPresent()`

ii) `int RemoteDebuggerPresent()`

iii) `bool DebugString()`

iv) `int CheckParentProcess()`

v) `bool` CloseHandleExcepDetection(HANDLE invalid)

vi) `bool` SingleStepExcepDetection()

However, you could use the rest of the methods before or after calling AntiAttachSelfDebug() in order to detect the presence of a debugger.
Those that will be used before calling AntiAttachSelfDebug() method will run at both parent and child processes.
Those that will be used after calling AntiAttachSelfDebug() will run only in the child process.

*****TIP***: You could use also the above methods with a little bit of imagination. In other words, since you know that your protected program will constantly be debugged by its parent process, you could use those methods in the opposite way after calling AntiAttachSelfDebug() so that if the child process is *not* debugged, then our code it is probably under attack.**

Initialize a new object and use the available methods

§ **DirectDebuggerDetection**

i) `bool DebuggerPresent()`

ii) `int RemoteDebuggerPresent()`

Include the **DirectDebuggerDetection.h** and add the **DirectDebuggerDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
DirectDebuggerDetection * directdbg = new DirectDebuggerDetection();
```

Use the methods:

```
if(directdbg->DebuggerPresent())
{

      cout << endl << "Attached Debugger Detected!!!" << endl;
}


else{

        cout << endl << "No Attached Debugger Detected..." << endl;

    }

if(directdbg->RemoteDebuggerPresent() == 1)
{

      cout << endl << "Attached Debugger Detected!!!" << endl;
}

else if (directdbg->RemoteDebuggerPresent() == 0){


cout << endl << "No Attached Debugger Detected..." << endl;

}
```

§ **IndirectDebuggerDetection**

i) `bool DebugString()`

ii) `int OpenServicesProcess()`

Include the **IndirectDebuggerDetection.h** and add the **IndirectDebuggerDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
IndirectDebuggerDetection * indirectdbg = new
IndirectDebuggerDetection();
```

Use the methods:

```
if(indirectdbg->DebugString())
{

cout << endl << "User-mode debugger through message to debugger has
been Detected!!!" << endl;

}

else{

  cout << endl << "User-mode debugger through message to debugger has
Not been Detected..." << endl;

}
```

```
if(indirectdbg->OpenServicesProcess()== 1)
{

  cout << endl << "Debugger Detected through OpenProcess to a system
process!!!" << endl;

}

else if(indirectdbg->OpenServicesProcess() == 0)
```

```
{

	cout << endl << "Debugger Not Detected through OpenProcess to a
system process..." << endl;

}
```

---

§ **WindowDebuggerDetection**

i) `bool SpecificWindowNameDetection(string windowname)`

ii) `bool SpecificWindowClassDetection(string classname)`

iii) `void SetListSize()`

iv) `int GetListSize()`

v) `bool ListWindowClassDetection(string * arraymemlocation , int listsize)`

Include the **WindowDebuggerDetection.h** and add the **WindowDebuggerDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
WindowDebuggerDetection * windowdebug = new WindowDebuggerDetection();
```

Use the methods:

```
//Detect Olly Debugger example through window class name

if(windowdebug->SpecificWindowClassDetection("OLLYDBG"))
{

	cout << endl << "Specific Window of Debugger/Reversing Tool
Detected through class name!!!" << endl;
```

```
}


else{

      cout << endl << "Specific Window of Debugger/Reversing Tool Not
Detected through class name..." << endl;
}


//Detect Olly Debugger example through window title


if(windowdebug->SpecificWindowNameDetection("OLLYDBG"))
{


      cout << endl << "Specific Window of Debugger/Reversing Tool
Detected through title name!!!" << endl;
}


else{

      cout << endl << "Specific Window of Debugger/Reversing Tool Not
Detected through title name..." << endl;
}


/*Detect a a debugger or a reversing tool from a predefined list
  of windows class names.
  Read the documentation above.*/


if(windowdebug->ListWindowClassDetection(windowdebug-
>SetReverseToolsList(),windowdebug->GetListSize()))
{


cout << endl << "A Window of Debugger/Reversing Tool has been
Detected through its class name from the predefined list!!!" << endl;

}


else{

cout << endl << "A Window of Debugger/Reversing Tool has NOT been
Detected through its class name from the predefined list..." << endl;

}
```

§ **ProcessDebuggerDetection**

i) `string * SetProcessList()`

ii) `void SetListSize()`

iii) `int GetListSize()`

iv)
```
int ProcessListDetection(string * arraymemlocation , int
  listsize)
```

Include the **ProcessDebuggerDetection.h** and add the **ProcessDebuggerDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
ProcessDebuggerDetection * procdbg = new ProcessDebuggerDetection();
```

Use the methods:

```
if(procdbg->ProcessListDetection(procdbg->SetProcessList(), procdbg-
>GetListSize()) == 1)
{

      cout << endl << "Debugger/Reversing Tool running process
Detected from our process name list!!!" << endl;
}

else if(procdbg->ProcessListDetection(procdbg->SetProcessList(),
procdbg->GetListSize()) == 0)
{
      cout << endl << "Debugger/Reversing Tool running process Not
Detected from our process name list..." << endl;

}
```

§ **ModuleDebuggerDetection**

i) `string * SetModulesList()`

ii) `void SetListSize()`

iii) `int GetListSize()`

iv) `int ModuleDetection(string * arraymemlocation, int listsize)`

Include the **ModuleDebuggerDetection.h** and add the **ModuleDebuggerDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
ModuleDebuggerDetection * moddbg = new ModuleDebuggerDetection();
```

Use the methods:

```
if(moddbg->ModuleDetection(moddbg->SetModulesList(), moddbg->GetListSize()) ==
1)
{

	cout << endl << "Debugger/Reversing tool detected through loaded
modules!!!" << endl;

}

else if(moddbg->ModuleDetection(moddbg->SetModulesList(), moddbg-
>GetListSize()) == 0)

{


	cout << endl << "No Debugger/Reversing tool detected through loaded
modules...." << endl;


}
```

§ **ParentProcessDetection**

i) `int CheckParentProcess()`

Include the **ParentProcessDetection.h** and add the **ParentProcessDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
ParentProcessDetection * ppdetect = new ParentProcessDetection();
```

Use the methods:

```
if(ppdetect->CheckParentProcess() == 1)
{

cout << endl << "Debugger/Reversing Tool Detected through parent
process id check!!!" << endl;


}

else if(ppdetect->CheckParentProcess() == 0)
{

cout << endl << "No Debugger/Reversing Tool Detected through parent
process id check..." << endl;


}
```

---

§ **CodeTraceTimeDetection**

i) `DWORD StartExecutionTime()`

ii) `DWORD EndExecutionTime()`

iii) `DWORD GetTimeLimit()`

iv) `DWORD GetTotalTime()`

v) `void SetStartTime()`

vi) `void SetEndTime()`

vii) `void SetTimeLimit()`

viii) `void SetTotalTime()`

ix) `bool IsCodeBeingTaced()`

Include the **CodeTraceTimeDetection.h** and add the **CodeTraceTimeDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
CodeTraceTimeDetection * tracetime = new CodeTraceTimeDetection();
```

Use the methods:

```
tracetime->SetStartTime();/* get the time before the execution of the
code block*/


/*the code block you want to check the execution time required goes
here*/


tracetime ->SetEndTime();/* get the time after the code block has
been executed*/


//perform the check

if(tracetime->IsCodeBeingTaced())
{

   cout << endl << "Debugger Detected through execution time
check!!!" << endl;

}


else{

      cout << endl << "Debugger Not Detected through execution time
check..." << endl;

}
```

§ HardwareBreakPointDetection

i) `int HwdBreakPoint()`

Include the **HardwareBreakPointDetection.h** and add the **HardwareBreakPointDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
HardwareBreakPointDetection * hwdbp = new
HardwareBreakPointDetection();
```

Use the methods:

```
if(hwdbp->HwdBreakPoint()==1)
{

	cout << endl << "Hardware Breakpoint has been Detected!!!" <<
endl;

}

else if(hwdbp->HwdBreakPoint()==0)
{

	cout << endl << "Hardware Breakpoint has Not been Detected..."
<< endl;

}
```

§ **ApiBreakPointDetection**

i) `int ApiBreakPoint(char * DLL, char * API)`

Include the **ApiBreakPointDetection.h** and add the **ApiBreakPointDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
ApiBreakPointDetection * apibp = new ApiBreakPointDetection();
```

Use the methods:

```
/* Example: Check for software breakpoint at the entrypoint of
OutputDebugStringA*/

if(apibp->ApiBreakPoint("kernel32","OutputDebugStringA") == 1)
{

      cout << endl << "Breapoint Detected on protected API!!!" <<
endl;

}


else if(apibp->ApiBreakPoint("kernel32","OutputDebugStringA") == 0){

        cout << endl << "Breapoint Not Detected on protected API..."
<< endl;

}
```

---

§ **SehDbgDetection**

i) `bool CloseHandleExcepDetection(HANDLE invalid)`

ii) `bool SingleStepExcepDetection()`

Include the **SehDebuggerDetection.h** and add the **SehDebuggerDetectionFunc.cpp** to your project.

Initialize a new instance of the class:

```
SehDbgDetection * sehdbgdetect = new SehDbgDetection();
```

Use the methods:

```
if(sehdbgdetect->CloseHandleExcepDetection((HANDLE)0x90909090)) //push an
invalid handle
{

	cout << endl << "Debugger detected through CloseHandle() exception!!!"
<< endl;
}

else
{

	cout << endl << "Debugger Not detected through CloseHandle()
exception..." << endl;

}
```

```
if(sehdbgdetect->SingleStepExcepDetection())
{

  cout << endl << "Debugger detected through Sigle Step exception!!!" << endl;

}


else{

	cout << endl << "Debugger NOT detected through Single Step exception..."
<< endl;

}
```

§ **AntiAttach**

i) `int AntiAttachSet()`

ii) `void AntiAttachSelfDebug()`

Include the **AntiAttach.h** and add the **AntiAttachFunc.cpp** to your project.

Initialize a new instance of the class:

```
AntiAttach * antiattach = new AntiAttach();
```

Use the methods:

```
if(antiattach->AntiAttachSet() == 1)
{

cout << endl << "Parent Anti-Attach has been set succesfully!!!" <<
endl;

}

else{

        cout << endl << "There was an error while setting the Anti-
Attach..." << endl;

}
```

```
antiattach->AntiAttachSelfDebug();/*see the documentation for more
info*/

cout << endl << "SelfDebug Anti-Attach has been applied!!!" << endl;
```

*****Final Note:*** Most of the methods will return an error code in case something goes wrong in the implementation.**
Check the documentation about the return values of each function and the corresponding possible error codes they may return.
Don't forget that the purpose of the detection methods is just to detect and not to provide countermeasures.
It is always up to you to decide what it should happen if a debugger is detected by using any of the detection methods provided here.

Always remember that imagination is the best anti-reversing method!