Exercise 4 – Hook, Hide and Anti-Debug

Deadline: 15.05.2013 13:37

Part 1 – Hook (35 points)

In this section you will hook keyboard messages in order to implement a simple user-level super-typer impersonator. Your end goal is to make notepad act as follows:

- Notepad should appear regular.
- Initially, typing characters into documents should work as usual.
- If the string "bazingaX" is entered (where X is a digit 0-9) into Notepad:
  - The string "bazingaX" will disappear.
  - A secret source input should be loaded into memory.
  - From now on, any printable character typed by the user should be hijacked and replaced with the next character from the source input.
  - The following will explain where the input text is taken from and how it will be used.
- Note that "bazingaX" is the exact order of keyboard strokes, meaning that "bazin", <backspace>, "ngaX" must not output a secret text.
- All strings are case-insensitive ("`bazinga0`" and "`BaZINga0`" should both work).

The source inputs will be hidden in alternate data streams within `Notepad.exe`.

Every NTFS file can have several streams. When you access, for example, `a.txt`, you actually access `a.txt::$data` (note the double colon). The full name for a stream is actually <filename>:<stream name>:<stream type>. The double colon in the previous example implies an empty stream name which means the default stream. You can append data and read data from alternate streams. Try the following commands:

```
echo "Hello World!" > a.txt
echo "Secret Message" > a.txt:secret
type a.txt
more < a.txt:secret
```

Note that the `type` command doesn't support streams, this is why the `more` command was used in the fourth line.

Find 10 interesting texts and append them to `notepad.exe` as streams named 0-9 (`notepad.exe:0` and so on). You may assume that all texts are no longer than 1024 characters – this will make your life easier later on.

Once, for example, "`bazinga7`" is typed into Notepad, you should load the contents of the stream `notepad.exe:7`. For every keystroke following this, you should replace the entered character with the next character from the source text until there are no more characters available. Once this happens, notepad returns to the initial mode and display keystrokes regularly.

Implementation details:

- You may assume that only a single instance of Notepad is running.
- In this part you'll need to hook the keyboard messages to Notepad. Read about SetWindowsHookEx.
- For the hook to work you'll need to implement a .DLL file. You should create an empty project and set its type to .DLL file. Read more about how to create .DLL files and about the DllMain function online. The hook procedure in your .DLL should be named `HookProc`. This is the exact name that should be exported by the .DLL (without name mangling). Use .def files for this purpose.
- Your .DLL must not stay loaded within arbitrary executables in the system. The hook mechanism will load your .DLL into many processes; make sure you only stay loaded within `Notepad.exe` (and `Explorer.exe` as explained in part 2).
- Your .DLL should be named `cptnhook.dll`.
- Note that you're not required to track mouse movements or arrow keys when capturing user's text input.
- When typing a secret message into Notepad, you may ignore the `'\r'` character to avoid double newlines.
- When Notepad is in the 'key hijacking' mode, there's no need to capture the 'bazingaX' sequence. Note that after the source input ended, you should continue to track the sequence as you initially did.
- Because you're writing non-console code, a good way to debug your programs is by using `OutputDebugString()`. You can view its output via an attached debugger or by using the very convenient DebugView utility.
- In order to test your code, write a small loader (an executable file) that uses `SetWindowHookEx` and your newly created .DLL file in order to implement the desired behavior. **No need to submit this loader** since it will change in part 2. This loader is just for making your life easier. Make sure your loader doesn't terminate, otherwise your hook will be removed from the system.

Part 2 – Hide (40 points)

The keen observer noticed that you've already started implemented hiding, when you placed the data into alternate streams. In this section you'll also try to hide the loader process.

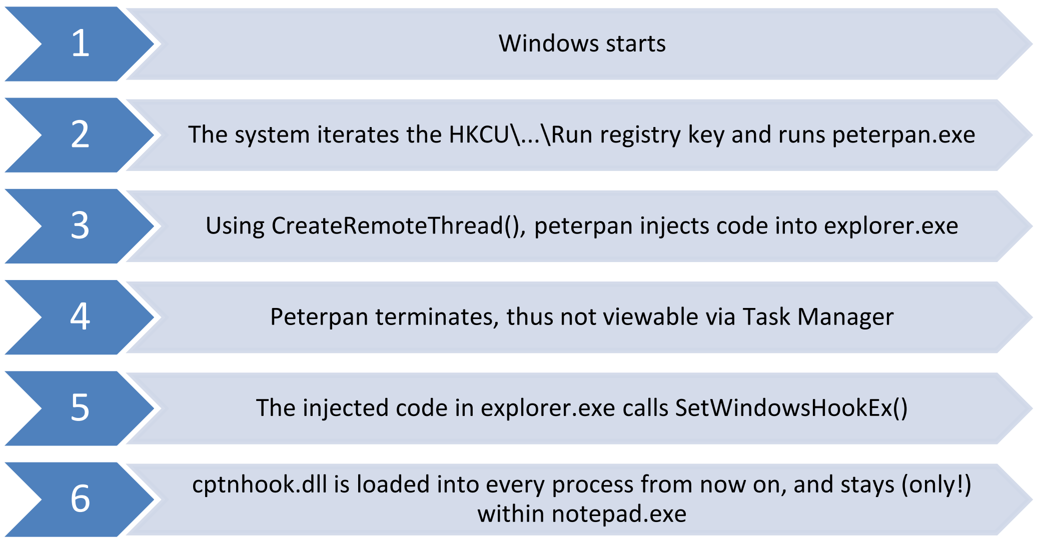
The previous part contains an implementation of a very nice hooking mechanism. The problem is that your loader is an executable which is easily visible when run. In this part you'll inject your code into `explorer.exe`. Your code will run as a thread within explorer – thus hidden. The loader from part 1 should be modified as follows:

- All previous functionality will be placed inside a function.

- That function's code will be injected into explorer.exe.
- The code will be executed as a thread within explorer.
- The code injection should be performed according to the process shown in class (slide "Via `CreateRemoteThread()` and `WriteProcessMemory()`").
- Note that you'll need to implement your function in a way that suits this method, as explained in class.

Also you'll make sure that your loader will be called after every reboot. Use the `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run` registry key. Add the name of your executable to this registry key manually, and make sure you indeed run after reboots. Note: **no need** to add yourself to this key programmatically, our tests will do that for you.

Here's a short diagram explaining the different parts of your implementation:



1. Windows starts
2. The system iterates the HKCU\...\Run registry key and runs peterpan.exe
3. Using CreateRemoteThread(), peterpan injects code into explorer.exe
4. Peterpan terminates, thus not viewable via Task Manager
5. The injected code in explorer.exe calls SetWindowsHookEx()
6. cptnhook.dll is loaded into every process from now on, and stays (only!) within notepad.exe

Implementation details:

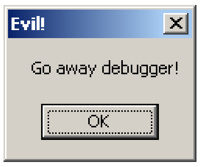
- Create a new executable named `peterpan.exe`. This executable will inject the functionality that you've implemented in your (temporary) loader in part 1 into `explorer.exe`.
- To prevent path issues, copy both files to the system directory (e.g. `C:\Windows\System32`), and make sure they run from there. We will test your code from that location.
- You may assume there's only one `explorer.exe` process available at the time of the injection, and that it never dies.

Part 3 – Anti-Debug (25 points)

Congratulations! Your hook now runs and it's even a bit hidden. In this part you'll implement anti-debugging tricks in order to make dynamic reverse engineering of your code harder.

Once your code detects that it's running within Notepad it should create an anti-debug thread which will do the following:

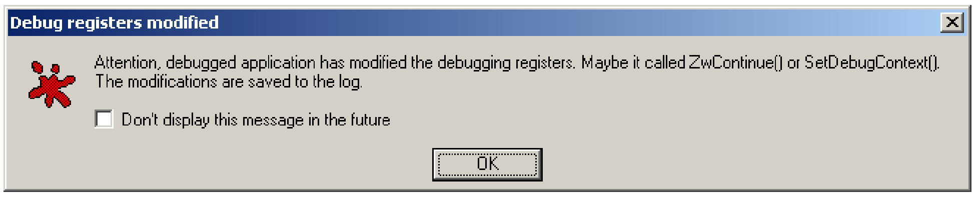
- Run in an endless loop. After each loop iteration, sleep for 5 seconds. Within each loop iteration:
  - Use `CheckRemoteDebuggerPresent()` to see if you're being debugged. If so create the following message box:



Evil!

Go away debugger!

  - Search for OllyDbg within the currently running process list. If you find it, display the above message box.
  - Use `OutputDebugString()` and `GetLastError()` to detect a debugger. If you find it, display the above message box. Find for yourself the correct way to use these two functions in order to detect a debugger.
  - Use `GetThreadContext()` and `SetThreadContext()` to change all writable debug registers to a value of 0. This should be done for either the anti-debug thread or the main thread, your choice. Note that OllyDbg is smart enough to detect this trick. You'll know that you've succeeded if the following message appears:



Debug registers modified

Attention, debugged application has modified the debugging registers. Maybe it called ZwContinue() or SetDebugContext(). The modifications are saved to the log.

Don't display this message in the future

- If a debugger was detected for 10 iterations in a row, terminate the process.

Your last anti-debugging trick will be a vanishing trick. You must make sure no readable strings appear in your code (both in the executable and in your .DLL). You'll only need to hide your own strings (for example, "`Go away debugger!`") and not system generated strings. The rule is, if it's a string that explicitly appears in your code, you must hide it. Use whichever method you want. Specify in the README file what you did.

Bonus (10 points)

In this section you'll have to supply an uninstall mechanism. If the string "`uninstallme`" is entered into Notepad, you should remove all traces of your program from memory and disk. Specifically:

- The hook should be unloaded.
- All injected code should terminate.
- All files and hidden streams must be deleted from the hard-drive.

- The startup mechanism should be disabled (delete the registry key – you may assume that no other `peterpan.exe` is registered within the run key).

All of the above should happen **instantaneously** and not after reboot. You might encounter problems with files being locked. Think how to handle these cases.

Submit your code as `bonus.tar`. Within `bonus.tar` include all your source files and `bonus.txt` that will explain your method and implementation.

Submission

All solutions should work under the Windows machines in Ross 19.

The following files should be included in the submitted tarball file:

- `cptnhook.h`
- `cptnhook.c`
- `cptnhook.def`
- `peterpan.h`
- `peterpan.c`
- `bonus.tar` (bonus only)
- README – Formatted according to the course guidelines.

A single TAR file named `ex4.tar` should be submitted. Make sure you run the pre-submit script before submitting the exercise.

Good luck!