SOME IMPORTANT POINTS

- This document is the second of a series of five articles relating to the art of hooking.

- As a test environment we will use an english Windows Seven SP1 operating system distribution.
When an inline hook is implemented it will overwrite the first bytes codes of a windows api in order to redirect code flow.

This kind of technique can be used in ring 3 or ring 0 modes.
CHARACTERISTICS

- Inline hooking is more robust than import address table (IAT) hooks.
- They do not have problems related to dll binding at runtime.
- They can trap any function call, not just system calls.
- No device driver is required.
- They are widely used in many renowned professional applications and well-known rootkits.
SOME GOOD UTILITIES

- At the **Defcon 17** Nick Harbour has presented a good tool named *apithief*.
  - The tool launches a process in a suspended state.
  - It then injects a DLL and hooks Win32 API functions.
  - The tool can then monitor the API behavior.
SOME BAD UTILITIES

- Well-known and widespread malware take advantage of this kind of technique to hook key API windows components in order to spy and steal sensitive information from the user.

- One of these famous malwares is Zeus, also known as Zbot, PRG, Wsnpoem or Gorhax.
Implementing an inline hook

- We inject our dll into the process we want to hijack. (Please see Userland Hooking in Windows)

- The first bytes of the target api are saved, these are the bytes that will be overwritten.

- We place a jump, call or push-ret instruction.

- The jump redirects the code flow to our function.

- The hook can later call the original api using the saved bytes of the hooked function.

- The original function will return control to our function and then data can be easily tampered.
WHAT ARE WE GOING TO OVERWRITE

- It is **very important to determine** what and where we are going to overwrite the bytes codes.

- Knowing exactly how much space we have at our disposal **is really important**, otherwise we can **destroy the logic** of the api, or write beyond his borders.

- An unconditional jump will require 5 bytes.

- We **need to disassemble** the beginning of the target api.
- The start of a Windows API function is usually the same.

- After XP-SP2 Microsoft decided to change the Windows API preamble:
  
  ```
  MOV EDI,EDI
  PUSH EBP
  MOV EBP,ESP
  ```

- This allows for hot-patching, inserting a new code without the need to reboot the box.

- We can use these five bytes to insert our own set of instructions.
WHERE ARE WE GOING TO OVERWRITE

- Another reason why we select the start of the function is because the more deeper into the function the hook is located, the more we must be careful with the code.

- If you hook deeper into the function, you can create unwanted issues.

- Things become more complex.

- Keep it simple and stupid.
AN UNSUCCESSFUL HOOK

- If we try to hook every function in the same way we can modify the logic of the function.

- In the picture below we have inserted an unconditional jump that destroy the logic of the function.
In this particular example the prolog of `httpsendrequest` API has been hooked with a jump to our function.

- We have used the first 5 bytes of the target function to place our `jmp` instruction.
In this practical example our target is a simple chat system using nc.exe, but every program using WS2_32.DLL!Send Api could be used.

We want to hijack the sent messages from one of two users.

To accomplish this task we inject our DLL into nc.exe process and we hook the aforementioned api.

Then we jump into our function and we change the stack parameters, then we adjust our stack to avoid incovenients and finally we jump again to WS2_32.DLL!Send api.

This hook example can be used to create more complex scenarios.
Before hooking the WS2_32.dll!send API we attach with our debugger to the nc.exe process which is already listening on port 9999. We can see the unhook prolog.
Server is listening for inbound connections (upper image) and the client connects (lower image). All is working normally and users are discussing without any issues.
At this time we **inject** our DLL into the *nc.exe* process.
Let's check using the our debugger how's things have changed.

The API preamble has been modified. It has been replaced with a jump to our evil function.
Let’s check our function at address 0x6FAC11D0.
The behavior of the chat has changed too :]

```
C:\>nc -vlp 9999
listening on [any] 9999 ...
connect to [127.0.0.1] from NEPTUNE.htbridge.local [127.0.0.1] 57099
Hey! How's life man? 
Very well: thank you.
Tell me how's your father?
Hmm, my father is doing well too :]
Pardon?
Sorry what?
What's up man?
Nothing all's right...
I do not understand you...
What? ??
```
Detecting inline hooks is pretty simple.

Rootkits detectors like *gmer*, or *BlackLight* from Fsecure do a good job.

Let’s do an injection test into *Internet Explorer 8.0 in Windows 7* and hook *wininet!HttpSendRequestW* API.
The hook is in place :]
But it has been successfully detected by gmer anti-rootkit.
**CONCLUSION**

- Inline hooks can be very useful when trying to reverse malware.

- Paradoxically nowadays malware use this technique to change the behavior of the operating system in order to steal personal information.

- By understanding and identifying hooks, thought, you will be able to detect most public rootkits and malwares.

- We hope this document help you understand and master inline hooks.

- Demo video: [Inline Hooking in Windows](#)
In future documents we will introduce Kernel Hooking.
REFERENCES

- Subverting the Windows Kernel (Greg Hoglund & James Butler)
- Professional Rootkits (Ric Vieler)
- Programming application for Microsoft Windows fourth edition (Jeffrey Ritcher)
- Programming Windows Security (Keith Brown)
- http://support.microsoft.com/kb/197571
- http://www.youtube.com/watch?v=2RJyR9igsdI
- http://www.gmer.net/
Thank you for reading

Your questions are always welcome!

Brian.mariani@htbridge.ch