Understanding and Defeating Windows 8.1 Kernel Patch Protection:

It’s all about gong fu! (part 2)

Andrea Allievi
Talos Security Research and Intelligence Group - Cisco Systems Inc.

aallievi@cisco.com

November 20th, 2014 - NoSuchCon

Who am I

- Security researcher, focused on Malware Research
- Work for Cisco Systems in the TALOS Security Research and Intelligence Group
- Microsoft OSs Internals enthusiast / Kernel system level developer
- Previously worked for PrevX, Webroot and Saferbytes
- Original designer of the first UEFI Bootkit in 2012, and other research projects/analysis

Agenda

0. Some definitions
1. Introduction to Patchguard and Driver Signing Enforcement
2. Kernel Patch Protection Implementation
3. Attacking Patchguard
4. Demo time
5. Going ahead in Patchguard Exploitation

Introduction

Definitions

- **Patchguard** or Kernel Patch Protection is a Microsoft technology developed to prevent any kind of modification to the Windows Kernel
- **Driver Signing Enforcement**, aka DSE, prevents any non-digitally signed code from being loaded and executed in the Windows Kernel
- A **Deferred Procedure Call**, aka DPC, is an operating system mechanism which allows high-priority tasks to defer required but lower-priority tasks for later execution
- An **Asynchronous Procedure Call**, aka APC, is a function that executes asynchronously in the context of a particular thread.

My work

- Snake campaign – Uroburos rootkit: an advanced rootkit capable of infecting several version of Windows, including Windows 7 64 bit
- Rootkit not able to infect Windows 8 / 8.1 because of security mitigations, enhanced DSE and Patchguard implementation
- Reversed the entire rootkit; this made me wonder how to to defeat DSE and Patchguard in Windows 8.1.
- This was done in the past with an UEFI bootkit - my approach now uses a kernel driver

Windows 8.1 Code Integrity

- Implemented completely differently than on Windows 7 (kernel 6.1)
- A kernel driver is usually loaded by the *NtLoadDriver* API function – ends in *ZwCreateSection.*
- A large call stack is made, that ends in *SeValidateImageHeader*
- *SeValidateImageHeader* - *CiValidateImageHeader* code integrity routine
- Still easy to disarm, a simple modification of the *g_CiOptions* internal variable is enough

Windows 8.1 Kernel Patch Protection

- If the value of the *g_ciOptions* variable changes, the Patchguard code is able to pinpoint the modification and crash the system
- Kernel Patch Protection implemented in various parts of the OS. Function names voluntarily misleading
- Patchguard in Windows 8.1 is much more effective than previous implementations
- Multiple PG buffers and contexts installed on the target system
- Uses a large numbers of tricks to hinder analysis

Windows 8.1 Kernel Patch Protection Implementation

Kernel Patch Protection – How does it work?

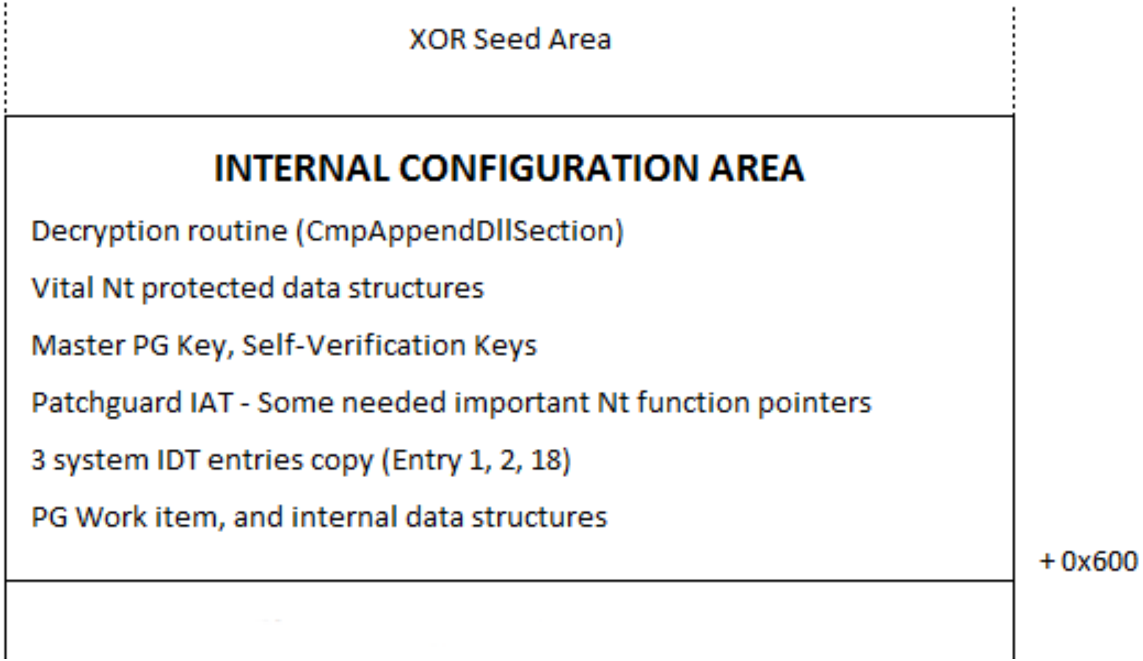
- *KeInitAmd64SpecificState* raises a Divide Error exception – execution transferred to *KiFilterFiberContext*
- *KiInitializePatchguard* is a huge function (~ 96 Kbyte of pure code) that builds a large PG buffer
- Structured Exception handling implementation: http://vrt-blog.snort.org/2014/06/exceptional-behavior-windows-81-x64-seh.html
- Other initialization point: *ExpLicenseWatchInitWorker* (rare)

```c
int KeInitAmd64SpecificState() {
    DWORD dbgMask = 0;
    int dividend = 0, result = 0;
    int value = 0;

    // Exit in case the system is booted in safe mode
    if (InitSafeBootMode) return 0;
    // KdDebuggerNotPresent: 1 - no debugger; 0 - a debugger is attached
    dbgMask = KdDebuggerNotPresent;
    // KdPitchDebugger: 1 - debugger disabled; 0 - a debugger could be attached
    dbgMask |= KdPitchDebugger;

    if (dbgMask) dividend = -1;      // Debugger completely disabled (0xFFFFFFFF)
    else dividend = 0x11;            // Debugger might be enabled

    value = (int)_rotr(dbgMask, 1);    // value64 is equal to 0 if debugger is enable
                                           // 0x80000000 if debugger is NOT enabled

    // Perform a signed division between two 32 bit integers:
    result = (int)(value / dividend);   // IDIV value, dividend
    return result;
}
```

The Kernel Patch Protection buffer

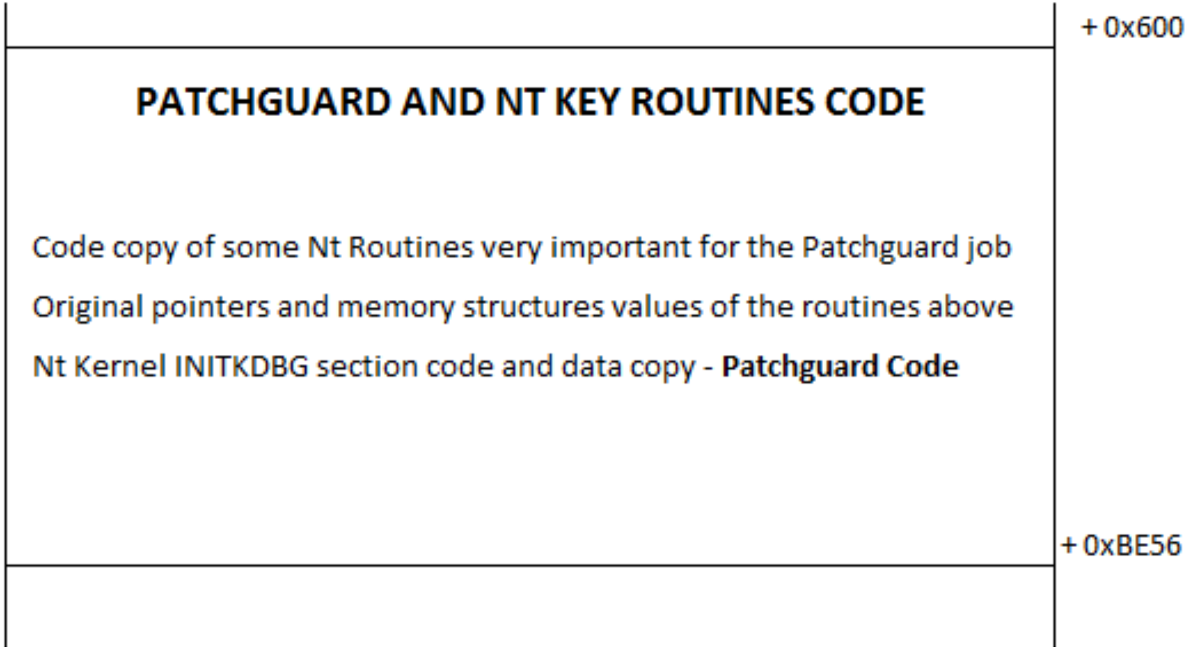
3 main sections surrounded by a random number of randomly generated values

1. Internal configuration area.



The Kernel Patch Protection buffer

2. Patchguard’s code and a copy of some NT kernel functions



The Kernel Patch Protection buffer

3. Protected code and data

+ 0xBE56

PROTECTED CODE AND DATA

16 Nt functions protected Code chunks descriptor and Key

Protected modules Exception directory Virtual address, size and Key

Huge array of Patchguard Keys containing one DWORD key for each code chunk described by a RUNTIME_FUNCTION structure of protected module Exception directory

Copy of data protected by Patchguard

Protected modules list: Nt kernel, Hal, WerLiveKernelApi, tm, clfs, pshed, kdcom, bootvid, ci, msrpc, ndis, ntfs, tcpip, fltmgr

+ 0x44E2C

XOR Seed Area

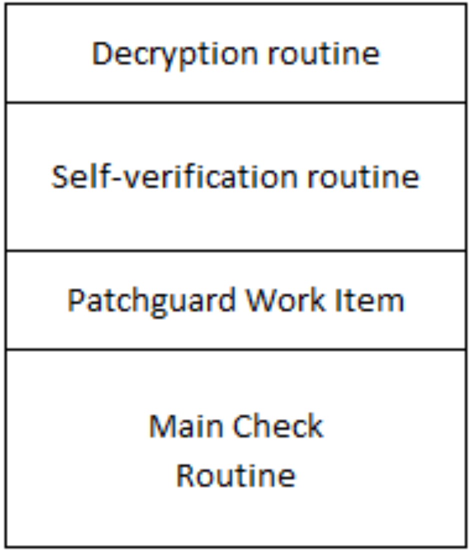
Implementation - Scheme

- Patchguard code is linked to the system in different ways: Timers, DPC routines, KPRCB reserved data fields, APC routines and a System Thread
- Patchguard initialization stub function *KiFilterFiberContext* **randomly decides** the PG link type and the number of PG contexts (1 to 4)
  - ✓ See here: http://blog.ptsecurity.com/2014/09/microsoft-windows-81-kernel-patch.html
- Entry points code: recover PG contexts, decrypts the first 4 bytes

cisco

Implementation – Scheme 2

- Patchguard code located inside the big buffer (section 2) organized mainly in 4 blocks:



Kernel Patch Protection – System checks

- Patchguard code implemented mainly in the “INITKDBG” section + chunks in “.text” section
- INITKDBG section copied, then discarded
- The self-verification routine executed with a copy of the original processor IDT
- Finally queues a Work item -> Main Check Routine…

The Main check routine

- Self-verification of the remaining bytes of section 1 and 2
- *PatchguardEncryptAndWait* function: on-the-fly encryption, waits a random number of minutes
- Verifies each code and data chunks of the protected kernel modules.
- Uses an array of Patchguard data structures
- If a modification is detected, a system crash initiated by “SdbpCheckDll” function

```
// Calculate a DWORD key for a specified Chunk
DWORD CalculateNtChunkPgKey(QWORD qwMasterKey, int iNumBitsToRotate, LPBYTE chunkPtr, DWORD chunkSize)
{
    // … some declarations here …
    for (count = 0; count < chunkSize / sizeof(QWORD); count++) {
        QWORD * qwPtr = (QWORD*)chunkPtr;  // Current buffer QWORD pointer
        qwCurKey = _rotl64((*qwPtr) ^ qwCurKey, iNumBitsToRotate); // Update the key
        chunkPtr += sizeof(QWORD);   // Update buffer ptr
    }

    // Calculate remaining bytes to process
    DWORD dwRemainingByte = chunkSize % sizeof(QWORD);
    for (count = 0; count < dwRemainingByte; count++) {
        LONGLONG qwByte =      // Current signed-extended byte
            (LONGLONG)(*chunkPtr);
        qwCurKey = _rotl64(qwCurKey ^ qwByte, iNumBitsToRotate);          // Update the key
        chunkPtr ++;           // Update buffer ptr
    }

    // Calculate DWORD key
    while (qwCurKey) {
        dwRetKey ^= (DWORD)qwCurKey;
        qwCurKey = qwCurKey >> 31;
    }
    // Keep in mind that the following key is verified after resetting its MSB: (dwRetKey & 0x7FFFFFFF)
    return dwRetKey;
}
```

Attacking Patchguard

Available attacks

All the available attacks have been defeated by the last version of Kernel Patch protection:

- x64 debug registers (DR registers)
- Exception handler hooking, Patching the kernel timer DPC dispatcher
- Hooking KeBugCheckEx and/or other kernel key functions
- Patchguard code decryption routine modification (McAfee method)

Available attacks – The Uroburos method

- Uroburos rootkit hooks *RtlCaptureContext* internal Nt Kernel routine.
- It's a function directly called by *KeBugCheckEx*, used by Patchguard to crash the system.
- Uroburos filters all the *RtlCaptureContext* calls made by *KeBugCheckEx*
- If the call is a Patchguard one, it restores the thread execution to its start address.
- If the IRQL too high - Uroburos exploits its own hook to *KiRetireDpcList*

Some new attacks

2 different types of feasible attacks idealized:

- Neutralize and block every Patchguard entry point
- On-the-fly modification of the encrypted Patchguard buffer, and make it auto-deleting

After my first article released, other guy, Tandasat method: hooking the end of *KiCommitThreadWait* and *KiAttemptFastRemovePriQueue* functions https://github.com/tandasat/PgResarch/tree/master/DisPG

Some new attacks – Can we innovate?

- All available methods try to prevent the Patchguard Code from being executed.
- Patchguard code can be an attacker best friend ☺

Forging Windows 8.1 Patchguard

My method uses a kernel-mode driver that does some things:

1. **Acquires all processors ownership** (very important step) and searches the Patchguard buffers starting from Windows Timers queue, DPC list, processor KPRCB structure, APC list, system threads list

2. Retrieves all the PG contexts (decryption key and so on...), and decrypts the Patchguard buffers

3. Analyses the buffer, retrieves all the needed information, and modifies it in a clever manner:

   ✓ Identify self-verify routine and disable it

   ✓ Identify main check routine and disarm it

   ✓ Let the Patchguard code execution continues

4. Re-encrypts Patchguard buffer, releases all processors ownership

```
; int __fastcall PatchguardWorkRoutine(LPVOID pgEncryptedBuff)
PatchguardWorkRoutine proc near

                sub     rsp, 48h
                call    PatchguardMainCheckRoutine
                lea     rcx, [rax+430h] ; RCX = PG code Protected FuncSect + 0x430
                mov     rdx, [rcx]
                or      rdx, rdx
                jnz     short CodeToEncrypt
                add     rsp, 40h
                push    rbx
                mov     rdx, rax        ; RDX = PG Buffer Base addr
                mov     rcx, [rax+400h] ; RCX = Pointer to the beginning of PG buffer
                mov     r8, [rax+0E0h]  ; R8 = ExFreePool Ptr
                mov     r10d, [rax+41Ch] ; R10 = InitKdbg Vsize + 600h + All functions total size
                mov     rbx, [rax+408h]
                mov     r9, rcx
                xor     r9, rsp
                lea     r11, PatchguardWorkRoutine

ReEncryptPgBuffQword:                   ; CODE XREF: PatchguardWorkRoutine+5A↓j
                xor     r9, [rdx]
                mov     [rdx], r9
                ror     r9, 3
                add     rdx, 8
                cmp     rdx, r11
                jb      short ReEncryptPgBuffQword
                 ........
                mov     rdx, rbx
                pop     rbx
                add     rsp, 8
                jmp     r8              ; JMP to ExFreePoolWithTag
PatchguardWorkRoutine endp

CodeToEncrypt:                          ; CODE XREF: PatchguardWorkRoutine+16↑j
```

Forging Windows 8.1 Patchguard - Details

The implementation is not easy. I have had to overcome some difficulties. Patchguard Contexts:

1. Timers – Search in system timer list
2. DPC – Search in system DPCs queue
3. APC – Insert an hook to KeInsertQueueApc
4. KPRCB – Analyse the undocumented fields in KPRCB structure (AcpiReserved, HalReserved)
5. Patchguard Thread – Search in the system threads list (very rare)
6. Other entry points (KiBalanceSetManagerPeriodicDpc) – KeInsertQueueDpc hook

Demo Time

Demo Time - Results

- Windows 8.1 Professional x64 – Fully updated
- Results:
  - ✓ Reliable method, works well on all versions of Windows 8.1
  - ✓ Hard to develop
- Comparison with other method:
  - ✓ Completely different method, platform dependent (it relies on “symsrv.dll” to obtain Windows symbols)
  - ✓ It can’t take advantage of Patchguard code to do some attacker’s dirty things ☺

Going ahead

Anti-Patchguard – Going ahead

- What happens if an attacker changes some verification hases directly located in the Patchguard buffer?
- A very strong weapon could bear:

  Use Windows 8.1 code to protect an attacker' rootkit code

- The Patchguard buffer, in its main section, includes 3 keys: The master key and 2 self-verification keys
- To achieve our goal we should modify some DWORD hashes, and finally we need to resign the entire Patchguard buffer

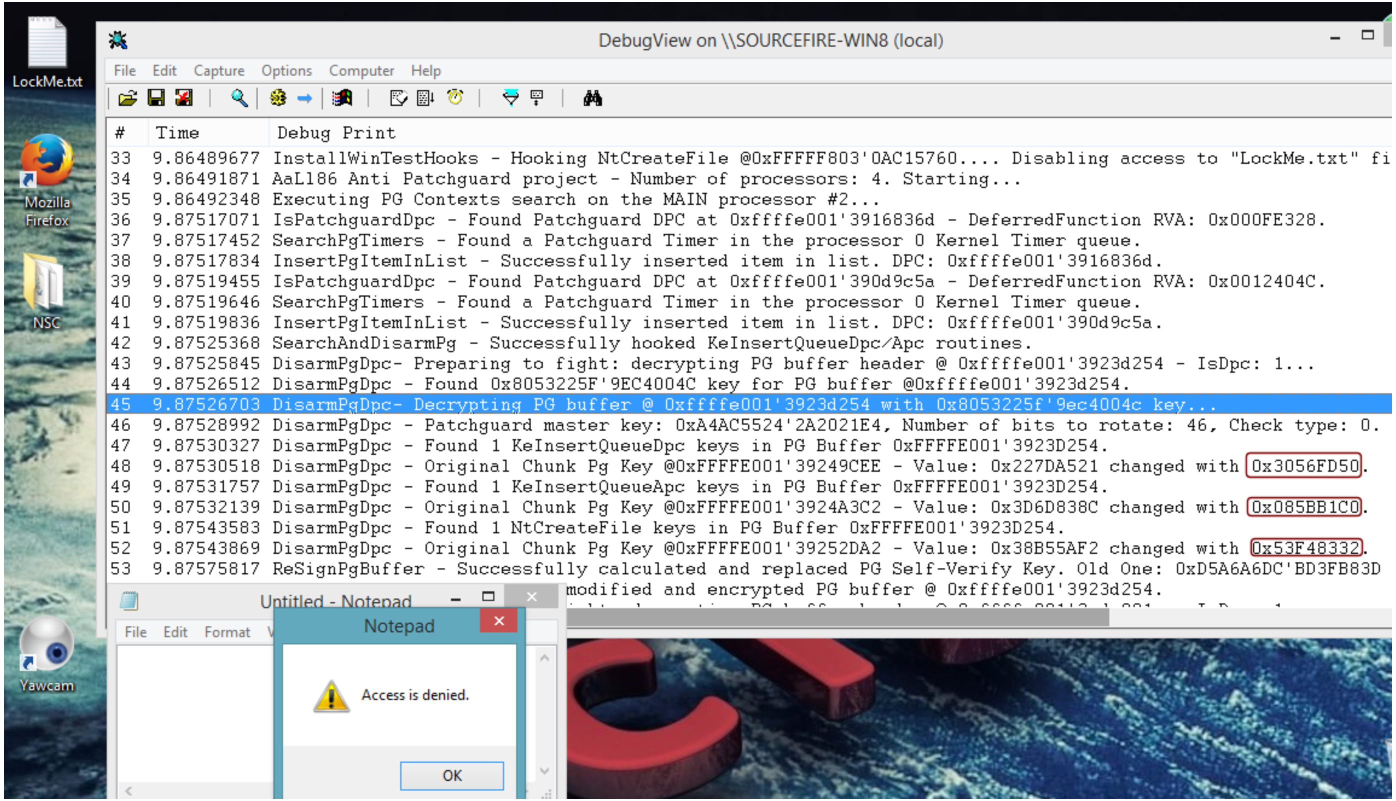
```
// Re-sign a Patchguard buffer modifying its Self-Verify keys
NTSTATUS ReSignPgBuffer(LPBYTE lpPgBuff) {
    // ... a lot of declarations here ...
    lpqwPgSelfVerifyKey = (QWORD*)((LPBYTE)lpPgBuff + 0x3F0);

    // Save original data and set to 0
    RtlCopyMemory(&orgPgWorkItem, pPgWorkItem, sizeof(WORK_QUEUE_ITEM));
    RtlZeroMemory(pPgWorkItem, sizeof(WORK_QUEUE_ITEM));
    qwOrgPgSignKey = *lpqwPgSelfVerifyKey; lpqwPgSelfVerifyKey[0] = 0;
    dwOrgNumOfVerifiedBytes = *lpdwNumOfVerifiedBytes; lpdwNumOfVerifiedBytes[0] = 0;

    // Now recalculate Patchguard Self-Verify Key
    qwNewSelfKey = CalculatePgSelfVerifyKey(qwPgMasterKey, iNumToRotate, (LPBYTE)lpPgBuff,
    dwNumBytesToSelfCheck);
    DbgPrint("ReSignPgBuffer - Successfully calculated and replaced PG Self-Verify Key. Old One:
    0x%08X'%08X - New One: 0x%08X'%08X.\r\n",
       qwOrgPgSignKey >> 32, (DWORD)qwOrgPgSignKey, qwNewSelfKey >> 32, (DWORD)qwNewSelfKey);
    *lpqwPgSelfVerifyKey = qwNewSelfKey;

    // Restore previous data
    RtlCopyMemory(pPgWorkItem, &orgPgWorkItem, sizeof(WORK_QUEUE_ITEM));
    *lpdwNumOfVerifiedBytes = dwNumBytesToSelfCheck;
    return STATUS_SUCCESS;
}
```

Use Windows 8.1 code to protect an attacker's rootkit code

- Our tests have demonstrated that the method is reliable, we have installed and protected a hook to the NtCreateFile API function
- Patchguard recognizes the new code as original and starts protecting it
- If an anti-rootkit solution tries to touch the "hook" code, the system suddenly crashes ☺
- Some problems, research still in progress
- Very **cool way** to recruit an opponent technology ☺ ☺
- Time for another demo?

cisco

Use Windows 8.1 code to protect an attacker' rootkit code



Questions Time

Resources and Acknowledgements

Available resources

Patchguard 8.1 Introduction material available on the VRT blog:

1. http://vrt-blog.snort.org/2014/04/snake-campaign-few-words-about-uroburos.html
2. http://vrt-blog.snort.org/2014/06/exceptional-behavior-windows-81-x64-seh.html
3. http://vrt-blog.snort.org/2014/08/the-windows-81-kernel-patch-protection.html

Analysis of previous versions of Patchguard:

1. http://www.zer0mem.sk/?p=271 (inspiration for my title)
2. http://www.uninformed.org/?v=3&a=3
3. http://uninformed.org/index.cgi?v=8&a=5
4. http://www.codeproject.com/Articles/28318/Bypassing-PatchGuard

Brand-new analysis, methods and techniques:

1. http://blog.ptsecurity.com/2014/09/microsoft-windows-81-kernel-patch.html
2. https://github.com/tandasat/PgResarch/tree/master/DisPG

Personal info

Andrea Allievi – **AaLl86**

Email: aallievi@cisco.com

Twitter: @aall86

Talos blog: http://blogs.cisco.com/talos

Sourcefire VRT blog (retired): http://vrt-blog.snort.org/

My personal website: www.andrea-allievi.com

Skype: aall86

For any question, information, send me a mail or a request on skype!

Acknowledgements - Thanks to

- TALOS Team for the help and support: Alain, Shaun, Angel, Douglas, Mariano, Emmanuel
- Microsoft engineers for developing a great technology
- My family and girlfriend for the support ☺
- Zer0mem for lending me the title ☺

Thank you for attending!

ps. Ready for the next Windows 10 Patchguard disarm?

CISCO

TOMORROW starts here.