Windows XP SP3 Registry Handling Buffer Overflow

by Matthew "j00ru" Jurczyk and Gynvael Coldwind
Hispasec

1. Basic Information

Name	Windows XP SP3 Registry Handling Buffer Overflow
Class	Design Error
Impact	Medium / High
Credits	Matthew "j00ru" Jurczyk, Gynvael Coldwind
Discovered	2008-12-20
Published	2010-05-29

2. Abstract

Microsoft Windows XP is a commonly used desktop operating system, released with Service Pack 3 at the time of writing this paper.
The vulnerable system component is heart of Windows NT family – the *ntoskrnl.exe* kernel executable itself. It is responsible for performing nearly all the critical system operations – most of the requests come from user-mode applications using the SYSENTER mechanism. System call set describes the functions available for a low-level application programmer – if one finds a buffer-overflow vulnerability in a syscall handler, it is very likely that he also will be able to run arbitrary code in the kernel's context (ring 0).

The vulnerability covered in this paper relies on the lack of user's input data validation in one of the Windows registry system call functions. The bug itself is present in an internal kernel function related to the "symbolic linking" mechanism, that provides a possibility to create invisible transition between a "fake" key (the forwarder) and the real ones. By crafting a properly malformed input as a function parameter, one is able to cause a pool-based overflow, consequently leading to a potential code execution and complete system compromitation.

The affected Microsoft Windows versions are all the systems prior to Windows Vista (including Windows XP SP3 with latest patches).

3. Vulnerability details

Before I start describing how the vulnerability itself works, I will shortly explain how the internal registry symbolic links work and how they are handled by the Windows kernel.
The entire Windows registry consists of a number of mixed .DAT files, each containing information about a separate registry tree. The binary format introduced by Microsoft decades ago has not been fully documented by the vendor himself, though many people decided to carry out some research on this topic. The results of their work are really promising – most of the format has been successfully reversed and put on downloadable websites. Since the format is being developed all the time as new requirements appear, the meaning of some flags and constant values being used in modern system is not yet known.
Each single registry key has its own structure inside the source REGF file. The structure contains every information that the system could possibly need during its work. The most important fields regarding a key record are:

- Key type bit mask
- Parent key record
- Point to sub-key list
- Pointer to security record
- Key name

This time, only the first field will be explained in detail. Although different sources claim that the, so called "Enumeration" field is a "static" value and can be set to only one predefined value, it's actually not true. It turned out that the following values can be mixed together, forcing the system to deal with a "malformed" key type, giving interesting results:

KEY_ROOT:	0x2C
KEY_LINK:	0x10
KEY_REGULAR:	0x20

So here is the way we can make one key point to another without even being "noticed":

- Put a link bitmask into the Type field of NK record
- Create a value named "SymbolicLinkValue" (this is a fixed name and cannot be changed!), of an internal type – REG_LINK (6) inside that key
- Put the destination path into the newly created value as a normal Unicode string.

After these actions are performed, the key should no longer exist as a normal record – every reference to a registry path containing a symbolic link is handled by the kernel so that the user-mode programs gets an open HANDLE to where the link points to, not to the forwarding key itself.

Under normal circumstances, the kernel does not use the registry contents during "critical" operations, but the link translation simply cannot be achieved without accessing the *SymbolicLinkValue* record. Having some essential knowledge about the registry internals, let's begin with the real vulnerability roots. It would be best for the reader to operate on the *Windows Kernel Research v1.0* sources since it shows the nature of the bug in a "clear form", but listings from both *IDA* and *WRK* will be presented below.

The buffer overflow is directly caused by the *SymbolicLinkValue* handling implementation. Most of the real job is performed by a non-exported function called *CmpGetSymbolicLink*. The function assumes that since the value is an Unicode string and is not generally used by any 3rd part applications (though some critical Windows components aim to use it for various purposes), its size is not going to be greater than 0xFFFF (65535d) bytes. The below code snippets should make everything clear:

```
if(!GetValueData(Buffer,&ValueLength))
{
  fail;
}

Length = (USHORT)ValueLength + sizeof(WCHAR);
```

(*vulnerable function pseudocode*)

```
PAGE:0049A071                 movzx   eax, word ptr [edi]
PAGE:0049A074                 lea     esi, [esi+eax+2]
```

Obviously, the highest 16 bits of the value length are ignored by the system parser, which could potentially lead to some kind of corrupted allocation and further code execution… Let's take a look at the rest of the function code:

```
NewBuffer = Allocate(Length);
if(NewBuffer == NULL)
{
  fail;
}
```

(*vulnerable function pseudocode*)

```
PAGE:004CA567 loc_4CA567:
PAGE:004CA567                 push    20204D43h       ; Tag
PAGE:004CA56C                 push    esi             ; NumberOfBytes
PAGE:004CA56D                 push    1               ; PoolType
PAGE:004CA56F                 call    _ExAllocatePoolWithTag@12
```

… and further…

```
CopyMemory(NewBuffer, Buffer, (DWORD)ValueLength);
```
(*vulnerable function pseudocode*)

```
PAGE:004CA57C                    mov     ecx, [ebp+var_8]
PAGE:004CA57F                    mov     [ebp+Destination.Length], cx
PAGE:004CA583                    mov     edx, ecx
PAGE:004CA585                    shr     ecx, 2
PAGE:004CA588                     mov     [ebp+Destination.MaximumLength],
si
PAGE:004CA58C                    mov     esi, [ebp+var_C]
PAGE:004CA58F                    mov     [ebp+Destination.Buffer], eax
PAGE:004CA592                    mov     edi, eax
PAGE:004CA594                    rep movsd
PAGE:004CA596                    mov     ecx, edx
PAGE:004CA598                    and     ecx, 3
PAGE:004CA59B                    cmp     [ebp+Source], 0
PAGE:004CA59F                    rep movsb
```

The case should need no explanation now. A 16-bit wrapped integer is used to allocate some pool memory, and then the other – real size of the data is used as *RtlCopyMemory* function parameter!

This typical kind of vulnerability creates a chance to achieve arbitrary code execution inside the *ExFreePool* kernel function – every user present on the system is allowed to create links without any restrictions, thus giving the vulnerability a “privilege escalation” status.

What should be noted is that the amount of bytes we use to overwrite the allocated memory area is not fully controlled by the attacker. To be more precise, it is not possible to say “An 8-byte overwrite is needed to gain control over the computer” and perform such attack – the vulnerability allows to overflow the buffer by any multiplicity of 0x10000. In other words, one can overflow the pool using at least ~65kB of junk data, which actually causes really serious damage to the drivers as well as the kernel module itself.

Another exploitation obstacle is an internal, so-called “deferred free” mechanism. In order to enhance the system performance, a special pending free requests’ list is kept in the kernel. Every time *ExFreePoolWithTag* function is called, a new record is being pushed on the list. When its size reaches a boundary value, it is being flushed by handling all the requests at once. Since triggering a pool overflow leads to most of critical kernel/driver data being irreversibly overwritten, there is no way the system could keep running correctly up to a moment when the real chunk deallocation is performed – one can either trigger code execution immediately after the overflow itself, either end up having the machine crashed inside some random device driver.

The author’s research implies that the “deferred free” mechanism is active on machines with at least ~512MB of physical memory. A relatively stable exploit has been developed for machines with lower amount of RAM. Unsuccessful or impossible (due to the above restrictions) exploitation process always leads to Denial of Service conditions, crashing the system.

4. Impact

The vulnerability allows an attacker to run arbitrary code with the system kernel (highest possible) privileges. It can be achieved using any kind of user account, both locally and remotely. The impact of a single attack, considering the above conditions, is rated as medium/high.

5. Disclaimer