Articles

KHOBE – 8.0 earthquake for Windows desktop security software

Published: 2010/05/05

In September 2007, we have published an article about a great disease that affected tens of Windows security products. The article called Plague in (security) software drivers revealed awful quality of kernel mode drivers installed by all the major desktop security products for Windows. The revealed problems could cause random system crashes, freezes and in some cases more severe security issues.

Today, we reveal even more serious problem of the Windows desktop security products that can be exploited to bypass a big portion of security features implemented by the affected products. The protection implemented by kernel mode drivers of today's security products can be bypassed effectively by a code running on an unprivileged user account. If you ever heard of SSDT hooks or similar techniques to implement various security features such as products' self-defense, we will show you how to bypass the protection easily.

Contents:

Introduction

The task of today's security software is to protect computers against malware and hacker attacks. This kind of application is generally very complex because it is trying to protect its users against threats of various kind. The security software use signatures and heuristic to detect known viruses, rootkis and trojan horses.

Malware writers are skilled enough to write malicious software that bypasses these detection techniques. To close this hole, modern security solutions implement HIPS and other forms of behavior control and blocking.

Because there were not many documented interfaces and APIs in the past which allowed to monitor and filter applications' activity in the system, many software vendors decided to make direct modifications to the user and kernel code and data structures. These modifications are often called as hooks. Today's security software often use tens of them to implement their security features.

Kernel mode hook implementation in security applications were subject of our research in 2007. This research showed that most of security software vendors implemented their kernel hooks very poorly and their applications were creating another holes into the operating system instead of protecting it. A special tool, BSODhook, was developed to find these vulnerabilities automatically. You can read more details about the research in Plague in (security) software drivers.

After we disclosed the results of our previous research, many vendors fixed the bogus implementation of their kernel mode hooks in reaction to the great publicity. This approach is required again because without the pressure from the media, it seems that many vendors are (again) absolutely uninterested in security holes in their software.

The goal of this paper

Most of security software perform their hooking on several well-known places. Some still base their protection mostly on user mode hooks which are, as reminded later in this document, faulty by design. Many vendors decided to alter the kernel part of system call mechanism implementation. They modify contents of System Service Descriptor Table (SSDT), which is often referred SSDT hooking. Other use different kind of kernel hooks for example by modifying the kernel code directly.

The main goal of this paper is to present an attack technique, called *the argument-switch attack* or *KHOBE attack*, that allows malicious code to bypass protection mechanisms of security applications. The attack is effective against user mode and kernel mode hooks. Because user mode hooks can be bypassed by simpler techniques, we focus on kernel mode hooks bypassing only. In the further text we demonstrate the attack techniques on SSDT hooks, which are the most common kernel hooks in today's security software. However, the attack techniques need no change to succeed against the other kinds of vulnerable kernel or user mode hooks.

For testing purposes, we have developed an engine called KHOBE (**K**ernel **HO**ok **B**ypassing **E**ngine) which simplifies writing of the exploits. With KHOBE we were able, in relatively short time, to verify that the presented vulnerability is an issue for many known security products on the market.

Original research

The research devoted to the problems we present in this article started in August 2008. Ten weeks later we finished it having a full documentation of the problem structured to three separate documents:

1. The documentation of the problem and explanations of its aspects on examples of

vulnerable implementations.
2. The documentation of possible solutions – how to implement all the requested functionality safely.
3. The documentation of KHOBE, the code library that can be used for fast and easy creation of exploits against the vulnerable implementations. This part was supported by various proof of concept exploits based on KHOBE.

The full results of the research were offered to our clients and other software vendors. In this article we use the parts of the first document to fully describe the problem. Other parts of the research will not be disclosed publicly and remain available for our business clients.

Why are user mode hooks insecure

Applications communicate with the operating system kernel through functions exported by various dynamic link libraries of Windows API, such as `kernel32.dll`, `user32.dll`, `advapi32.dll` and `ntdll.dll`. These routines usually transmit execution flow to the lower layer of the interface libraries. The layer on the bottom, represented by `ntdll.dll` (and partially `user32.dll` and `gdi32.dll`), uses special instruction, such as `INT 0x2E`, `SYSENTER` or `SYSCALL`, to call the kernel.

Various security applications place their hooks on the beginning of Windows API routines to stop malicious code working. However, all dynamic link libraries belonging to Windows API reside in the user mode portion of process' address space, hence the application code might avoid calling them which effectively bypasses hooks made by security software. If the application needs to communicate with the kernel, it can use the system call instruction directly. And this action cannot be caught or prevented by any type of user mode hooking.

SSDT hooking

System Service Descriptor Tables belong to the kernel mode part of system call interface implementation. They contain addresses of routines (referenced also as system services) that user mode code can invoke indirectly as a result of the special system call instruction. Through SSDTs, one can control every useful transition from user mode to kernel mode. That is why they seem to be so suitable for implementation of real time protection and self-defense.

Security software often modifies addresses stored in the tables to point to its own routines, called hook functions or hook handlers. These routines perform various checks to find out whether the call endangers the system or might be a threat to the security application itself. In these cases, the hook handler blocks the call or asks user what to do. If the hook routine considers the call as safe, it invokes original system service with same parameters it got from the user mode application. This must be done without changing the context of the call. Everything must be transparent to the original system service which expects no installed hooks.

The following code snippet shows a sample hook handler for **NtLoadDriver** system service. The handler attempts to block every attempt to load driver which service name is "SampleDriver".

```
00 NTSTATUS NewNtLoadDriver(IN PUNICODE_STRING DriverServiceName)
01 {
02   NTSTATUS status=STATUS_SUCCESS;
03
04   if (KeGetPreviousMode()!=KernelMode)
05   {
06     WCHAR *buffer=NULL;
07
```

```
08      __try
09      {
0A        ProbeForRead(DriverServiceName,sizeof(UNICODE_STRING),1);
0B
0C        SIZE_T name_len=DriverServiceName->Length/sizeof(WCHAR);
0D        ProbeForRead(DriverServiceName->Buffer,name_len*sizeof(WCHAR),2);
0E
0F        SIZET_T buf_len=(name_len+1)*sizeof(WCHAR);
10        buffer=(WCHAR*)ExAllocatePoolWithTag(NonPagedPool,buf_len,POOL_TA(
11        if (buffer)
12        {
13          RtlZeroMemory(buffer,buf_len);
14          RtlCopyMemory(buffer,DriverServiceName->buffer,name_len*sizeof(
15
16          WCHAR *drv_name=wcsrchr(buffer,L"\\");
17          if (drv_name) drv_name++;
18          else drv_name=buffer;
19
1A          SIZE_T drv_len=wcslen(drv_name);
1B
1C          PWSTR pattern=L"SampleDriver";
1D          SIZE_T pat_len=wcslen(pattern);
1E
1F          if (pat_len==drv_len)
20          {
21            UNICODE_STRING pattern_name;
22            RtlInitUnicodeString(&pattern_name,pattern);
23
24            UNICODE_STRING driver_name;
25            RtlInitUnicodeString(&driver_name,drv_name);
26
27            if (RtlCompareUnicodeString(&pattern_name,&driver_name,TRUE)==
28              status=STATUS_ACCESS_DENIED;
29          }
2A        } else status=STATUS_INSUFFICIENT_RESOURCES;
2B      } __except (EXCEPTION_EXECUTE_HANDLER)
2C      {
2D          status=STATUS_INVALID_PARAMETER;
2E      }
2F
30      if (buffer)
31          ExFreePoolWithTag(buffer);
32
33      if (!NT_SUCCESS(status))
34          return status;
35    }
36
37    status=OldNtLoadDriver(DriverServiceName);
38
39    return status;
3A  }
```

Example 1: Vulnerable NtLoadDriver hook handler.

Firstly, the hook handler routine determines the origin of the call (line **04**). Filtering is applied only to calls coming from user mode. SSDT hooks are not very suitable for protection against kernel mode threats and hence passing calls that come from kernel mode is very common in hooking drivers. Then, the routine jumps to the **try**/**except** block where the verification of the parameter is made.

The hook routine uses **ProbeForRead** function (lines **0A** and **0D**) to verify that the address of the UNICODE_STRING parameter *DriverServiceName*, and the address specified in the *Buffer* member of the structure, are pointing to user mode memory. If the application passed kernel mode address to the **NtLoadDriver** system call, **ProbeForRead** raises exception which the hook handler catches by the **try**/**except** block and indicates by setting status code to STATUS_INVALID_PARAMETER on line **2D**.

If no exception occurs, the hook handler copies the name of the driver service registry key to the kernel memory and compares it with the "SampleDriver" string, which stands for malicious driver service, that should be blocked from loading. The **try**/**except** block catches all exceptions that can be raised during copying of the *DriverServiceName* structure, which might happen when the application specified address in user mode which points to invalid memory region. You can read more about checking parameters of various types in Plague in (security) software drivers.

The comparison of unicode strings is made on line **27**. If the handler discovers that the application is attempting to load the malicious "SampleDriver" driver, it blocks the call by setting status code to STATUS_ACCESS_DENIED on line **28**.

If the call originates from kernel mode or the verification finishes successfully, the hook handler invokes the original system service and passes the original *DriverServiceName* pointer as its argument (line **37**). Then it returns status code of the call (line **39**). Passing original values of parameters to the system service is very important because its code will also check their validity using **try**/**except** block and **ProbeForRead** function. The hook handler must behave in a way that does not impact the execution flow of the original function.

The problem

The code of system services runs on IRQL PASSIVE_LEVEL which also applies to their hook handlers. Code running at this level can access pageable memory and, when the scheduler decides, might be preempted by another thread. And the scheduler plays the key role in the argument-switch attack.

The hook handler might be preempted at every moment of its execution. Imagine that the context switch occurs just before the line **37** is about to be executed. At this point, all security checks are done and the hook handler is about to invoke the original system service. Before the scheduler switches back to the hook handler, many threads might use the processor for some time, including other threads of the application that called **NtLoadDriver**.

If another thread of the application is scheduled onto the processor, it might theoretically change the driver service name passed to the **NtLoadDriver** call because the whole string lies in user mode portion of address space. The execution of hook handler had advanced too far to detect this modification and perform additional checks to stop malicious driver from loading. When the scheduler switch thread context back to the thread executing the hook handler, the original system service is invoked and might load driver which name was not subject of any security check made by the security software. And this is exactly how the argument-switch attack works.

Things get more simple on multiprocessor systems where two or more threads of the same application might run really simultaneously and the context switch at the crucial moment is not really needed. Today, multiprocessors or multi-core processors are very common hardware in desktop computers.

In the following sections, the thread that invokes the system service will be called *the attacker thread* and one that attempts to modify the parameters in order to bypass security checks will be referenced as *the faker thread*. Surprisingly, the attacking technique may be implemented even if the attacker thread runs in different process than the faker thread. In further text, however, we limit our thoughts to a single attacking process only.

Argument-switch attack to bypass checks of security software

As shown in the previous section, the idea of the argument-switch attack is quite simple. The attacker calls the system service with values of parameters that will certainly pass the

checks made during the execution of services' hook handlers. When the faker thread gets its time slice, it tampers the contents of the parameters to values that would never pass the checks of the hook handler. If the tampering operation occurs after the security checks are done but before the original service, which does the main work, is called, the attack is successful.

So, one of the remaining questions is, which types of parameters can be faked and how to achieve it. Generally speaking, there are three types of arguments passed to system services:

- integer values, boolean values, bit masks (access mask, flags etc.) and other unstructured arguments;
- pointers to flat buffers or more complex structures such as UNICODE_STRING or OBJECT_ATTRIBUTES;
- handles to kernel objects.

Parameters of the first type cannot be tampered by the attack – during the system call, they are copied to the kernel stack, which is not accessible from user mode.

The idea of faking pointers was described for the example code of **NewNtLoadDriver** function in the previous sections. The faker thread cannot modify the value of the pointer, because it is copied to the kernel stack, hence not accessible from user space. However, the pointer refers to a region in the user memory and contents of this region can be changed. If the change occurs in the right time (after the security check but before the execution of the original system service), the hook is bypassed.

The situation with handles is more difficult. Again, the value of the handle itself can not be changed because it is copied to the kernel stack. What we can do, however, is to change the actual meaning of the handle value. We will describe the attack on the example of sample **NtTerminateProcess** hook. The handle type can be understood as a special type of pointer (handles refer to kernel objects like pointers do to memory regions), so one can conclude that the attack on handle arguments will have much in common with tampering pointers.

Consider the following implementation of **NtTerminateProcess** hook handler. We assume that the security software allows us to get handle with PROCESS_TERMINATE flag to the protected process and that the level of defense lies only in this **NtTerminateProcess** hook which blocks attempts to terminate process with PID saved in *protected_process_id* variable.

```
00 NTSTATUS NewNtTerminateProcess(IN HANDLE ProcessHandle, IN NTSTATUS Exi
01 {
02   NTSTATUS status=STATUS_SUCCESS;
03
04   if (ProcessHandle && (ProcessHandle!=NtCurrentProcess) && (KeGetPrevi
05   {
06     PROCESS_BASIC_INFORMATION info;
07     status=ZwQueryInformationProcess(ProcessHandle,ProcessBasicInformat
08     if (NT_SUCCESS(status))
09     {
0A       ULONG pid=info.UniqueProcessId;
0B       if (pid==protected_process_id)
0C         status=STATUS_ACCESS_DENIED;
0D     }
0E   }
0F
10   if (NT_SUCCESS(status))
11     status=OldNtTerminateProcess(ProcessHandle,ExitStatus);
12
13   return status;
14 }
```

Example 2: Vulnerable NtTerminateProcess hook handler.

Unlike the **NtLoadDriver** hook example, validating of parameters does not require any probing or copying, actions that must be performed within **try/except** block. There is no

need to check *ExitStatus* argument, because every value is valid as the process exit code, and the most of the required handle checking (*ProcessHandle* parameter) is performed in **ZwQueryInformationProcess** call. The hook handler only checks the origin of the call and whether the handle could be valid or does not point directly to current process. This validation is performed on line **04**.

If the handle seems to be valid and the call comes from user mode, the hook handler routine tries to find out which process the handle belongs to. Basic information, which includes PID of the target process, is retrieved on line **07** via call to **ZwQueryInformationProcess**. This routine filters out handles of bad type or entirely invalid.

The final check is made on line **0B** where the PID of the target process is compared to PID of the process protected by the security software. In case of equality, the return status code is changed to "access denied" (line **0C**) and the original handler is not invoked. Otherwise, the original system service is called (line **11**). It is left for the original service code to verify that the handle was opened with PROCESS_TERMINATE access right, which is required for terminating the process via this service call.

Similarly to the **NtLoadDriver** example, the success of the attack depends whether the faker thread is able to alter the meaning of the *ProcessHandle* parameter when the attacker thread is executing lines from **08** to **10**, i.e. after the check of PID, but before the execution of the original code. It seems that there is not much space for the attack, but in fact even the first instructions of the function called on line **11** can be executed. All instructions in the original function prior the handle value passed in *ProcessHandle* is dereferenced can be executed without affecting the result of the attack. Moreover, it should be noted that unlike our sample hook handler, hook handlers in security products are usually very complex and there are many instructions to be executed between the security check and the original function call.

The faker thread can not change the value of the *ProcessHandle* parameter, because it is pushed onto the kernel stack before the hook handler is executed. However, it can change the meaning. This is analogous to pointers – it is not possible to change their values but it is possible to change the memory region they point to. Changing meaning of a handle is just a bit more difficult than changing the contents of memory region.

The faker thread must destroy the handle passed by the attacker to the system call. This operation must be performed after the hook handler successfully returns from **ZwQueryInformationProcess** call, but before the original **NtTerminateProcess** is executed. Then the faker thread opens a handle to the protected process with PROCESS_TERMINATE access right (which will be allowed by security software due to our assumptions made in the beginning). If we are lucky, the new handle will have the same value as the closed one. In case the new handle is created before the attack thread reaches line **11** and invokes original **NtTerminateProcess**, the attack succeeds and the protected process is terminated. This sounds like a very unlikely thing to happen, but with a good implementation, such as KHOBE, it is possible to effectively, i.e. in real time, attack pointer arguments as well as handle arguments.

Combined attack

The previous sections illustrate how to bypass a single SSDT hook like **NtLoadDriver** or **NtTerminateProcess**. However, some security products implement multi-level protection – for example, they hook **NtOpenProcess**, where all attempts to get handle with PROCESS_TERMINATE access right are blocked. Additionally, **NtTerminateProcess** is intercepted to block termination attempts of those, who somehow obtained the handle with sufficient access rights.

However, even two-level protection is not enough and can be bypassed by slightly modified variant of the argument-switch attack. We will describe its semantics on the protection scheme presented in the previous paragraph. We need one attacker and two faker threads. During the initialization phase, a handle to the current process is opened with access rights that prevent its use for process termination. Then the attacker thread invokes **NtTerminateProcess** with this handle. This call is intercepted by the **NtTerminateProcess**

hook handler. The hook handler finds out that the handle does not belong to any of the protected processes. Hence, the termination operation is allowed and the original system service is about to be executed.

Now, we need to use the standard argument-switch attack to bypass the hook of **NtOpenProcess**. If this sub-attack finishes before the attacker is scheduled on the processor again, the attack succeeds and the protected process is terminated.

At this moment, the attack needs scheduler to switch to the first faker thread which closes the handle to our process opened during the initialization phase and tries to call **NtOpenProcess** with PID of the current process again. The task of the second faker thread is to change the CLIENT_ID argument to point to the protected process just after the **NtOpenProcess** hook handler invokes the original system service. So the scheme of the sub-attack is similar to the attacks described in previous sections – the first faker thread plays role of the attacker thread and the second one acts as the faker thread.

If the sub-attack successfully finishes before the **NtTerminateProcess** system service begins its execution and if the handle to the protected process has the same value as the handle opened during the initialization, the combined attack succeeds and the protected process is terminated.

If the single argument-switch attack was seemed to be a very unlikely thing to happen, the combined version seems to be a miracle. Using KHOBE, however, we have proved again that on multiprocessor hardware the combined attack can be performed again in real time, the combined attack needs only few seconds to succeed.

Table of vulnerable software

We have performed tests with today's most Windows desktop security products. The results are presented in table below. The results can be summarized in one sentence: If a product uses SSDT hooks or other kind of kernel mode hooks on similar level to implement security features it is vulnerable. In other words, **100 % of the tested products were found vulnerable**. The only reason there are not more products in the following table is our time limitation. Otherwise, the list would be endless.

Product name and version	Result
3D EQSecure Professional Edition 4.2	VULNERA
avast! Internet Security 5.0.462	VULNERA
AVG Internet Security 9.0.791	VULNERA
Avira Premium Security Suite 10.0.0.536	VULNERA
BitDefender Total Security 2010 13.0.20.347	VULNERA
Blink Professional 4.6.1	VULNERA
CA Internet Security Suite Plus 2010 6.0.0.272	VULNERA
Comodo Internet Security Free 4.0.138377.779	VULNERA
DefenseWall Personal Firewall 3.00	VULNERA
Dr.Web Security Space Pro 6.0.0.03100	VULNERA
ESET Smart Security 4.2.35.3	VULNERA
F-Secure Internet Security 2010 10.00 build 246	VULNERA
G DATA TotalCare 2010	VULNERA
Kaspersky Internet Security 2010 9.0.0.736	VULNERA
KingSoft Personal Firewall 9 Plus 2009.05.07.70	VULNERA
Malware Defender 2.6.0	VULNERA
McAfee Total Protection 2010 10.0.580	VULNERA
Norman Security Suite PRO 8.0	VULNERA

Product name and version	Result
Norton Internet Security 2010 17.5.0.127	VULNERA
Online Armor Premium 4.0.0.35	VULNERA
Online Solutions Security Suite 1.5.14905.0	VULNERA
Outpost Security Suite Pro 6.7.3.3063.452.0726	VULNERA
Outpost Security Suite Pro 7.0.3330.505.1221 BETA VERSION	VULNERA
Panda Internet Security 2010 15.01.00	VULNERA
PC Tools Firewall Plus 6.0.0.88	VULNERA
PrivateFirewall 7.0.20.37	VULNERA
Security Shield 2010 13.0.16.313	VULNERA
Sophos Endpoint Security and Control 9.0.5	VULNERA
Trend Micro Internet Security Pro 2010 17.50.1647.0000	VULNERA
Vba32 Personal 3.12.12.4	VULNERA
VIPRE Antivirus Premium 4.0.3272	VULNERA
VirusBuster Internet Security Suite 3.2	VULNERA
Webroot Internet Security Essentials 6.1.0.145	VULNERA
ZoneAlarm Extreme Security 9.1.507.000	VULNERA

Final observations and notes

The argument-switch attack requires specific behavior from system scheduler which is impossible to ensure from user mode. However, this fact does not affect the strength of the attack too much. If the goal of the attack is to perform the action that the security software always blocks, the attacker and faker threads might try the method many times till they succeed. Our experiments show that in most cases, the attack succeeds after few attempts and sometimes even the first attempt is successful. The number of attempts needed can be radically lowered using smart manipulation of the threads' priorities.

The attack pattern described in this document does not, in general, use any feature present only on privileged user accounts. This means, that successful attack might come even from processes running under restricted user account.

We showed how the argument-switch attack works on the example of SSDT hooks. However, this method might be used to exploit other types of hooks as well. It only needs several conditions, which hold in most common implementations of hooks. The attack might function also on hooks, which:

- run on low IRQL with scheduler enabled, this condition theoretically is not needed to be true on multiprocessor systems, where the attacker and the faker threads can run on different processors simultaneously;
- accept handles to kernel objects, or pointers referencing memory in user mode portion of address space;
- perform validity and security checks;
- call original routines with unchanged values of the important arguments.

During our research we also focused on how to defend against the argument-switch attack. We realized, that securing kernel hooks might be quite complicated task for security software vendors, particularly for those which software uses huge amounts of SSDT and other types of hooks. This paper does not disclose our solutions of the problem.

One might think that installing two security applications will protect against the argument-switch attack. This makes, however, the situation even worse. Imagine that two security applications, *A* and *B*, hooked the **NtOpenProcess** system service. When application tries to open a process, the execution flow is intercepted by the hook handler installed by driver of application *B* (this application hooked the service when it was hooked already by

application *A*).

If the hook handler of *B* is not implemented securely against the argument-switch attack, it gives no additional security to hook routine of *A*. It might even create additional security risk for *A*. The hook handler of *B* might call the hook handler of *A*, but it does not have to; if *B* calls original system service instead, the function of operating system is not affected (except for performance), but *A* does not have a chance to perform its security checks.

On the other hand, if *B* wants to preserve the functionality of the *A*'s hook handler and calls it with original arguments then *B*'s hook handler is vulnerable to the argument-switch attack regardless the implementation of *A*'s hook handler. So, unless *A* and *B* are implemented perfectly and can rely on each other's perfection (which is a very strong assumption similar to cooperation of two applications written by competitive vendors), *A* or *B* will be vulnerable to the argument-switch attack. This is why installing more than one security product on a single machine is a risk today even if their vendors claim the products to be compatible.

Summary

This paper presents attack pattern called the argument-switch attack which shows that common implementations of kernel mode hooks are not secure. This attack represents serious threat because many security software vendors base their security features on hooking. We tested the most widely used security applications and found out that all of them are vulnerable. Today's most popular security solutions simply do not work.