Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik IV

Seminar of Advanced Exploitation Techniques, WS 2006/2007

hacking in physically addressable memory

a proof of concept

by ***David R. Piegdon***

April 12, 2007

Supervisor: Lexi Pimenidis

The author, student of computer science at the *Rheinisch-Westfälische Technische Hochschule Aachen*, may be reached via email. To obtain the full source code, this paper or other information, please refer to the following link.

David Rasmus Piegdon
david.rasmus.piegdon@rwth-aachen.de
http://david.piegdon.de/products.html

This text was written on *Linux* with *vim*, typesetted with LaTeX and generated by an implementation of *RFC2795*. Please send fruit to the author. No bananas.

Abstract

Several advances in hacking via DMA will be introduced; attacks to steal ssh private keys, inject code and obtain an interactive shell via firewire only will be presented.

All of these advances are based on data structures that are required by the CPU to provide virtual address spaces for each process running on the system. These data structures are searched and then parsed to solve the puzzle of randomly scattered pages in the physical memory, thus being able to read and write in each processes virtual address space.

Most of the attacks introduced will be adaptable to all kinds of operating system and hardware combinations, but as a sample target, Linux on an IA-32 system with the kernel-options `CONFIG_NOHIGHMEM` or `CONFIG_HIGHMEM4G`, `CONFIG_VMSPLIT_3G` and `CONFIG_PAGE_OFFSET=0xC0000000` is used.

Contents

1 Introduction

Several advances in hacking via DMA will be introduces. As a foundation for the attacks, `libphysical` and `liblinear` will be introduced to access virtual address spaces of any process on the to-be-attacked host.

To have a simple, generic interface for all kinds of physical memory sources, we implemented `libphysical`, where backends for new memory sources may be plugged in. It includes backends for IEEE1394 and filedescriptors so far.

In **section 2**, `libphysical` will be introduced and different backends with their advantages and disadvantages will be discussed.

Once, access to the physical memory of a system is obtained, there are two obvious ways to extract useful information from it:

- It is possible to parse the operating systems internal data structures holding all relevant information about loaded drivers, running processes et al.
- It is possible to use the information that the operating system provides to the hardware to tell it about the virtual address spaces of each process.

The first scenario will not work between different operating systems and architectures, it will be necessary to write a parser for each combination of them, possibly even for different versions of the same operating system.

The latter uses an information structure that only changes between different architectures, as the architecture relies on it. Furthermore there is a well defined algorithm for using this information (implemented in hardware in the architecture, but well defined in the reference manuals for this architecture, so system designers can provide valid data to the hardware). On the other hand, the first approach will give much more information about the system than the second, as we obtain all information directly from the kernel structures, while using the second approach we only can enter virtual address spaces of processes.

In [Bur06], an approach is introduced that parses kernel-structs of windows and Linux kernels. In the following *we* will use the second approach for most attacks, as this seems to be more robust and automatable. As the above mentioned paper is about *finding an attacker* and not *attacking*, the forensic personal does know about the architecture, the operating system and version and can build a copy of the system to test its tools, while an attacker only can guess the architecture and operating system and needs more robust tools for his attacks. (Obviously this is only a short-term argument, as an attacker can also write such tools for *all* known OS version and architecture combinations).

In **section 3**, `liblinear`, an interface to access virtual address spaces, will be introduced. It incorporates a backend for IA-32[1] and functions to find virtual address spaces.

In **section 4**, several attacks will be introduces, ranging from simple information gathering up to obtaining an interactive shell via DMA only.

In **section 5**, prospects will be given, what further kinds of attacks seem to be possible and/or may be of interest.

[1] This backend is missing algorithms for less-used operation-modes of the IA-32 architecture, but it will work at least for most kinds of the *Linux* kernels ($\leq$ 4GB RAM). It has not yet been tested with *Windows* or *MacOS X* and is missing features (Virtual-8086 mode) to work with DOS-processes running inside Windows.

2 Physically addressable memory sources: libphysical

Typically, processes and users are never granted access to physically addressed memory, as this addressing mode circumvents any protection methods provided by virtual addressing to separate processes from each other and the operating system. Only the operating system may use physical addressing to prepare address spaces for each running process, manage these, access special memory of extension cards and alike, or even only during bootstrapping as Linux does. But there are several ways of obtaining access to physical addressed memory. As stated, physical addressing will circumvent protection mechanisms. Thus access to it should require system administrator rights or physical access to the hardware of the underlying system.

To have a simple, generic interface for all kinds of physical memory sources, we implemented a simple interface, where backends for new memory sources may be plugged in. This interface is called `libphysical` and includes so far backends for IEEE1394 and filedescriptors.

Modern computer hardware provides many protection and memory management mechanisms in hardware. This includes hardware to provide a virtual address space for each process, protection mechanisms to restrict a process to its own resources only, paging to extend memory to harddisks and fragment available memory, caching to access frequently used memory faster, and more. Obviously, all these features are architecture and operating-system dependent. An interested reader may read documentation on system programming (e.g. [S+02] or [Int06d, Int06e]) to obtain further information.

Assume a process with its virtual address space and its corresponding set of pages. Each page in this virtual address space may be:

- a real, physical memory page that is mapped into the virtual address space, possibly cached in the CPU's cache,
- a used page that is swapped to other media, like a harddisk
- (depending on the operating system) a mapped buffer or file
- not used, and thus not mapped

Swapped pages will and mapped pages may be loaded only on demand (i.e. when the process tries to access the page), as access to a non-mapped page by a process will generate a page fault and the operating system then may map the demanded page. Access to completely unused pages, via this mechanism, will create the well known segmentation fault.

When one obtains access to physically addressed memory, that is a set of pages, each page may be a page of memory of a random process, a buffer page of a process, a page used by the operating system (kernel code, kernel data, kernel stack, IO buffer, ...), an unused page or a page used to give the CPU information on how to handle virtual addresses, as this is done in hardware. The latter pages will be called address translation tables; for more information on these, see section 3.1.

2.1 Generic problems: swapping, multiple accessors, caching, address translation

As stated, virtual pages of a process may be swapped, buffers may only be created on demand, and pages may be cached in other memory.

As physical memory only gives access to pages that are mapped from this physical memory, we will be unable to access swapped pages and buffers that have not been mapped. There is no simple solution to access this data; it is required to call special operating system routines to do this; but as access to physical memory does not include access to the CPU by itself, and these routines may be different from operating system to operating system, there is no simple solution.

Depending on the method used to access the memory, a parallel accessor may be using the same memory at the same time. E.g. when using firewire (see 2.2) to read a page of a currently running task, this task may access, thus read or write this page at the same time. For instance, if both CPU and another accessor write at the same moment to the same address location, it is undefined, which write access will be performed and thus override the other one (actually it depends on timing and caching, but we have no way of knowing, when the systems CPU accesses a page and if it is cached). Also, reading and writing at the same time may be impossible via the given method; thus many atomic commands used for process synchronization, like "test and set", will not exist. Caching may also prevent these.

If a page is cached in another, faster memory, a copy of it will typically reside in physical memory. In most cases we will not know if a page is cached or not; on IA-32 however the address translation tables contain a flag for each page telling the CPU if it may be cached or not. Depending on the way used to access the memory, it may circumvent the cache or not have access to it at all. When accessing a page, changes made by a task running in parallel may not be visible to us immediately and changes made by us may be invisible to a parallel task or maybe even overwritten by the cache at any time. Special care needs to be taken to minimize this risk. When writing to pages, only choose pages that are not cached or unlikely to be cached while writing; when reading pages, consider that the data may change at any time or may have changed yet.

As stated above, on systems using paging, physical memory will mostly be a concatenation of random pages, each one either used by a random process or the operating system. A minor part of these pages will be address translation tables, telling the CPU what the virtual address space of different processes looks like. Where these pages are is only known to the operating system and the CPU. For a detailed discussion, see section 3.1.

2.2 IEEE1394

IEEE1394, also known as firewire (Apple) or iLink (Sony), is an extension bus available on many modern computer systems and devices. In contrast to USB, which is a serial periphery bus, firewire is a high-speed serial expansion bus with features like guaranteed bandwidth (which is of interest for many real-time applications, like media crunching), DMA[2] and the ability to connect multiple nodes with a single firewire-bus. The concept of bus master and bus slaves, as known from USB, is only virtual. Typically when plugging together a firewire bus, a node is randomly selected to be the master and manages this

[2]Direct Memory Access

bus. Most of these nodes have the ability to be bus master.

DMA is implemented in hardware by the OHCI chip set; it is used to release the CPU from I/O operations. Mechanisms of preventing unwanted access exist, but many drivers do not activate these methods by default. In the case of windows, the operating system can be tricked into giving DMA to any device (see section 2.2.2).

Up to 64 devices may be plugged into one bus. Each one will choose a bus-unique node ID [0..63]; the bus master will have the highest node ID. All node IDs will be sequential, starting with 0. Access to memory of a node will require a 10 bit field for the bus ID, a 6 bit field for the node ID and a 48 bit address field. On Linux, `libraw1394`[3] provides an easy and portable interface to do this. Ideally, the full physical memory is accessible via the 48 bit address field, mapped in this address space starting from `0x0`; around `0xffff f000 0000` several control state registers (CSRs) are located that provide information about the given target node and the capabilities of the firewire device.

To disable the DMA-feature of firewire completely, load the *ohci1394* module with option *phys_dma=0* on Linux or set the security mode in open-firmware to something different than "none" for apples powerbooks and ibooks.

For more information on the underlying hardware or protocols, please refer to [Pro00, And99] or the `libraw1394`[3] documentation.

2.2.1 Physical security, attacks with embedded devices like iPod

Designing protection schemes of security systems should always include taking care of physical security, even if the object of interest is a pure virtual one (like a secret key). In this case, physical security, i.e. the restriction of access to the underlying hardware of a running computer system, is often forgotten and thus weak.

In general, a professional attacker will most likely choose the weakest link of security for his attacks. If he finds that going straight to the computer and stealing it, or plugging an embedded device like the iPod into it is easier than other known attacks, he will try it.

Some typical physical attacks on computer systems are:

- booting custom operating system
- opening the case and attacking the hardware directly
- stealing the whole system or its harddisks
- installing cameras, microphones and key-loggers
- installing specially crafted PCMCIA cards

Firewire does not require to boot a custom operating system, to open the case, steal parts of the hardware, install any hardware (except plugging in the firewire device) or specially crafted hardware. Access via firewire is as easy as plugging in a firewire device, like the iPod, letting it do its job and unplugging it.

On the other hand, plugging in firewire devices will generate syslog messaged on Linux systems.

[3] `libraw1394`: http://www.linux1394.org/

2.2.2 Problems with IEEE1394 DMA, solutions

Using firewire to access the memory of a running system *may* generate non maskable interrupts (NMI) on the target system and even leave the operating system (Linux 2.6.17) in a non-usable state. There are some laptops and workstations with on-board controllers being disturbed by this, sometimes resulting in crashes. Computers with PCI extension card, though, just worked fine during all tests. It is possible that the deeply incorporated on-board controllers interfere with the CPU accessing the main memory. This needs to be investigated.

Using `libraw1394`, it is possible to read different blocksizes of data via firewire. It seems like different hardware does allow bigger blocks to be read at different addresses. 4 byte blocks should always work; 1024 byte blocks may be read with some hardware, if the address resolves to the physical memory. Control state registers are likely to be readable only in 4 byte blocks.

Windows XP does use OHCI features to implement protection mechanisms to prevent random devices from reading any memory location. Adam Boileau,"TMASKY" and others have shown [Boi06] that, by pretending to be a device like an iPod, which "deserves" DMA (in terms of marketdroid[4]-logik), it is possible to circumvent this "protection" and to trick Windows into giving an attacker DMA. This attack is as simple as reading an iPod's config rom from its CSR and using `libraw1394`'s `raw1394_update_config_rom()` to use the copy. Adam Boileau has implemented a simple script to do this. We have written our own tool in C using `libraw1394`, which is located in `1394csrtools/`.

2.3 Filedescriptor: /dev/mem, memory dumps

Another source for physical memory may be given to an attacker via a filedescriptor. This filedescriptor may refer to a memory dump or the Linux `/dev/mem` device. In case of a plain memory dump, many of the mentioned problems lapse: no caching will be performed, no concurrent process will change the dumped data. In the case of a filedescriptor referring to `/dev/mem`, other accessors will exist, as `/dev/mem` is referring to the systems memory; caching on the other hand should not be a problem as we are not circumventing any caching system (like the CPU), but directly using it.

2.3.1 Problems with /dev/mem, solutions

`/dev/mem` on Linux has one other, major problem: On 32 bit systems, the virtual address space has a size of 2^{32} bytes, i.e. 4GiB. The virtual address space of a Linux process is divided into a userspace-part, where code, libraries and stack are mapped, and a kernelspace-part, where the complete kernel, data structures and the kernel-stack of this process are located. The userspace thread will not be able to access the kernelspace pages; but when the userspace process calls a system call, the CPU will jump into a different protection level and execute the system call entry code, which is part of the kernel. At this time the kernel runs in the same virtual address space, just at a different protection level. Default kernel configs divide the 4GiB virtual address space into a 3GiB userspace-part (`0x0000 0000 - 0xbfff ffff`) and a 1GiB kernelspace-part (`0xc000 0000 - 0xffff ffff`). Different kernel config options (`CONFIG_VMSPLIT_3G`, `CONFIG_PAGE_OFFSET`) can change the split ratio, but the splitting itself is a basic design decision and thus a requirement for Linux to work. In the upper kernelspace-part, the lower physical memory will be fully mapped

[4] see jargon-files, http://catb.org/esr/jargon/html/M/marketroid.html

(up to about 900MiB), the kernel will - in this case - be located at roughly `0xc100 0000`, i.e. at the physical address `0x0100 0000`. During kernel configuration, these 900MiB are called "precious lowmem", because it is the only memory accessible by the kernel in a simple manner. When allocating kernelspace structures - this includes all address translation tables [5] - the kernel will normally only use this lowmem. The `/dev/mem` driver will access this lowmem, when a userspace process utilizes it. But if the process tries to read more than the mapped lowmem, `/dev/mem` will fail to return this, as it is not mapped in the process. Thus, when using `/dev/mem`, only approx. 900GiB lower physical RAM will be accessible.

It may be possible to work around this restriction: As stated, all address translation tables are located in the accessible memory. A process could try to find its own pagetable and manipulate it to map other regions of physical memory into its userspace section; then, no further access to `/dev/mem` would be required. As the reader will see in section 3.3, identifying processes in Linux can mostly be as easy as looking at the upper userspace stack page. Research on a system with 1.5GiB RAM has shown that most of these pages will be in the unmapped area, thus many of the processes will not be identifiable by this method. To make its own pagedirectory identifiable from the others, a process may e.g. map a random file at a typically never used address (e.g. `0x6000 0000`), make sure the mapping was successful, access this file (to prevent a missing mapping-on-demand) and then search for the signature in all found pagedirectories. When found, it may map this pagedirectory at a special location and directly manipulate it. It should be taken care of the problem that a manipulated pagedirectory will only be reloaded into the CPU after a re-scheduling of the process, but a simple `usleep()` should suffice. The author is not sure, under which circumstances the Linux kernel does manipulate a process's pagedirectory; but obvious things like mapping new regions or unmapping mapped regions by system calls should be avoided, as the system may overwrite the manipulated pagetable with a new, adjusted one.

2.4 Other sources

The ideas described in this paper should be easily adoptable to all memory sources giving access to physically addressable memory, this may include e.g. remote management cards, suspend-2-disk images or virtual machines that have an interface to access the virtual machines' memory. `qemu` is such a virtual machine, providing a `gdb` remote stub to attach a debugger.

To use a new physical source with the methods introduced in the later sections, it is only required to write a new backend for `libphysical`.

[5] on IA-32, these are: pagedirectories, pagetables, local- and global descriptor tables

3 Translating virtual to physical addresses

3.1 Basic address translations

All multitasking environments that fulfill current requirements have to provide virtual address spaces for each running process or thread. For performance and security reasons this address translation from a processes virtual address to an address valid in physical memory is normally performed in hardware. These mechanisms can include e.g. segmentation and paging.

A normal process's memory is divisible into several blocks or *segments*: the *code segment* contains all the code that may be run; the *data segment* contains the static data that is known at compile time, global structures or deliberately allocated memory (including the heap); the *stack segment* contains the stack, including local variables. On some architectures, it is possible to assign segment descriptors, referring to defined memory regions, to segment registers. This assignment will influence the further behaviour of address translation: all addresses will from there on be taken to be relative to the bound of the memory region specified by the segment descriptor. This mechanism is called *segmentation*.

Paging will divide the virtual address space of a process into several consecutive *frames* of a specific page-size (typically 4096 bytes). Virtual addresses can be split into frame number and frame offset; the frame number is translated (mapped) via a translation table into a physical page number and the frame offset is used as an offset into this physical page. If a frame does not have a corresponding physical page, it is called to be unmapped. Unmapped pages can be non-existing pages or can e.g. be swapped to slower media like harddisks.

For a detailed description and discussion of these two important mechanisms, the reader may refer to a course on system programming, e.g. [S+02].

`liblinear` provides a software solution for address translation. The provided interface is similar to `libphysical`; it needs a physical memory source (in form of a `physical_handle`), and information about the target architecture. It provides some functions to find address translation tables in the raw memory and functions to use them to access the induced virtual address space.

3.2 Example implementation: IA-32 backend for `liblinear`

On the IA-32 architecture, the CPU can run in various modes of operation; for modern multitasking operating systems the *protected mode* is the preferred one. The protected mode can use a two-level address translation: first it will translate the *logical address*, consisting of a segment selector (which is an index into either the local or the global segment descriptor table) and an offset to the *linear address*. The linear address is then translated via paging to the *physical address*. (The paging translation is optional and needs to be enabled by setting a special flag in a control register of the CPU.)

A Linux process runs in a simple 4GiB flat virtual address space; no segmentation is required. Thus, Linux will create (among others that are not of interest for us) four special segments during boot-up: for each privilege level (i.e. kernelspace and userspace), it will create segments for both code and data. These four so called *flat* segments will span the full virtual address space of 4GiB, thus effectively eliminating segmentation. The address of the *global descriptor table*, holding the description of these segments, is then loaded into the *global descriptor table register* (GDTR) and the specific segment registers are loaded

with segment selectors referring to the segments[6].

The IA-32 architecture divides the 4GiB virtual address space into 1024 4MiB-frames. This splitting is defined by the *pagedirectory*. Each entry of a pagedirectory is 4 bytes long, thus the pagedirectory is $4 \cdot 1024 = 4096$ Bytes long. Each of these *pagedirectory entries* (PDEs), if present (its `PRESENT`-flag is set), can either refer to a 4MiB physical page or a pagetable dividing this virtual 4MiB frame further into 4KiB frames. A *pagetable* is again consisting of 1024 4-byte *pagetable entries* (PTEs), each corresponding to a 4KiB frame.

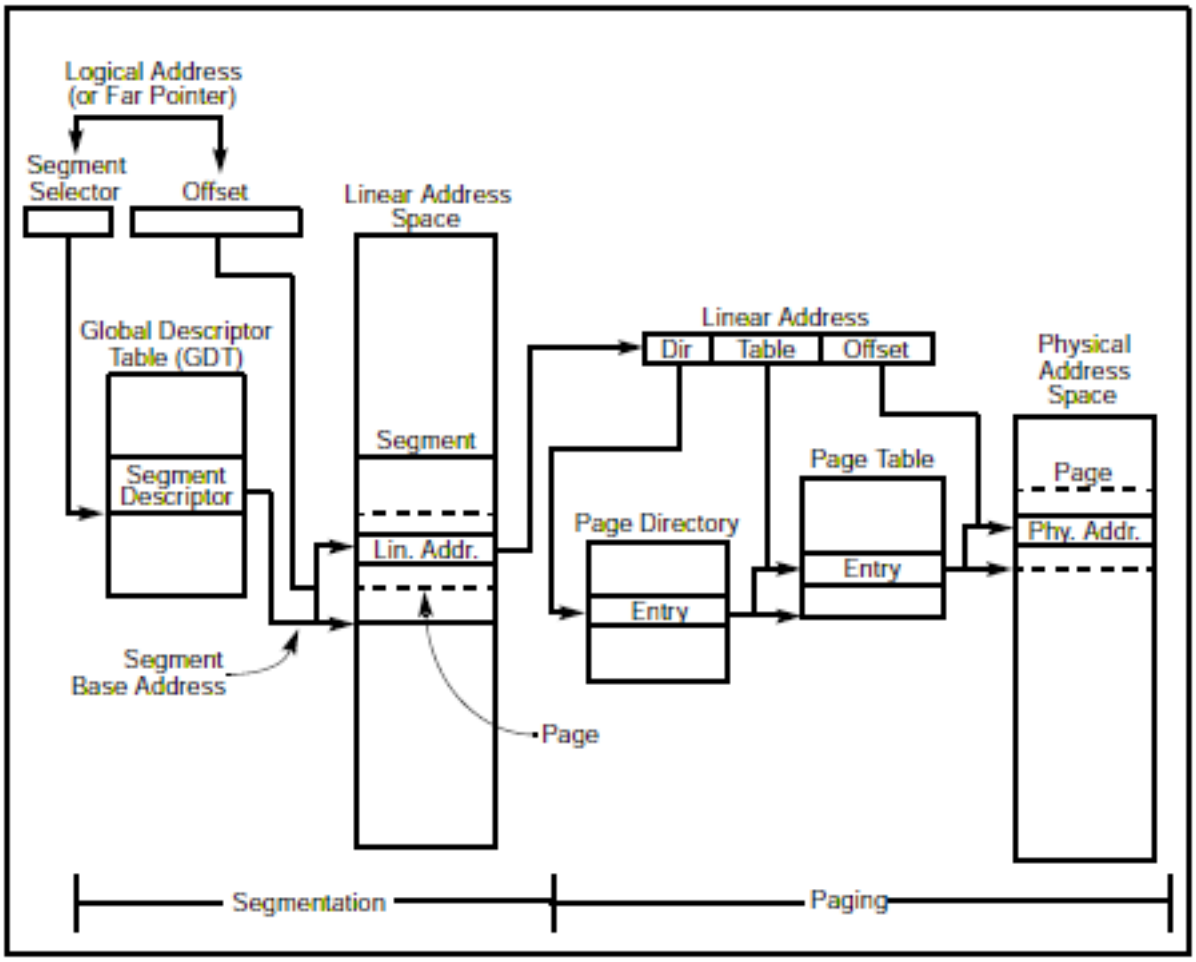

Figure 1: IA-32 Segmentation and Paging process (image taken from [Int06d])

As newer IA-32 CPU features like *36 bit page size extension* (PSE-36) and *physical address extension* (PSE) are not used in case of the proposed circumstances[7], their reflection is omitted here. Furthermore it is not always possible to know from the physical memory only, if these features are enabled. A sample-installation of a system to be attacked should give these informations. Also, PAE and PSE-36 are not yet implemented in `liblinear`. PSE though (not PSE-36) is enabled with the given options (and implemented), as one can determine by the use of 4MiB-pages.

[6] This initialization is done in `linux/arch/i386/kernel/head.S`, GDTs are defined at symbols `boot_gdt_table` and `cpu_gdt_table`

[7] `CONFIG_NOHIGHMEM` or `CONFIG_HIGHMEM4G`, `CONFIG_VMSPLIT_3G`

For an extensive documentation of the IA-32 architecture one should refer to the *Intel 64 and IA-32 Architectures Software Developer's Manual* ([Int06a, Int06b, Int06c, Int06d, Int06e]), especially [Int06d].

3.3 Finding address translation tables

When accessing a range of memory via physical addressing, it is necessary to find address translation tables to make sense out of the vast, unsorted number of pages. Typically, translation tables are not marked as such and as we can not access the processor or the operating system to ask, where these are, we have to search them. The following methods have proven themselves when being combined: for all pages: make a simple test if a page *could* be a pagedirectory (3.3.1) and if so, analyse this page in detail (3.3.2).

3.3.1 OS and architecture dependencies; typical address space layout

Obviously, address translation tables are architecture and operating system specific; but within an architecture and an operating system, they will often share data or specific patterns that are identifiable. For instance, when searching for Linux IA-32 address translation tables, one can omit searching the segment descriptor tables (see section 3.2) and concentrate on finding pagedirectories. There are several special patterns that can be found in a typical pagedirectory of a Linux process running on IA-32. Following is a layout of the typical virtual address space of a userspace process:

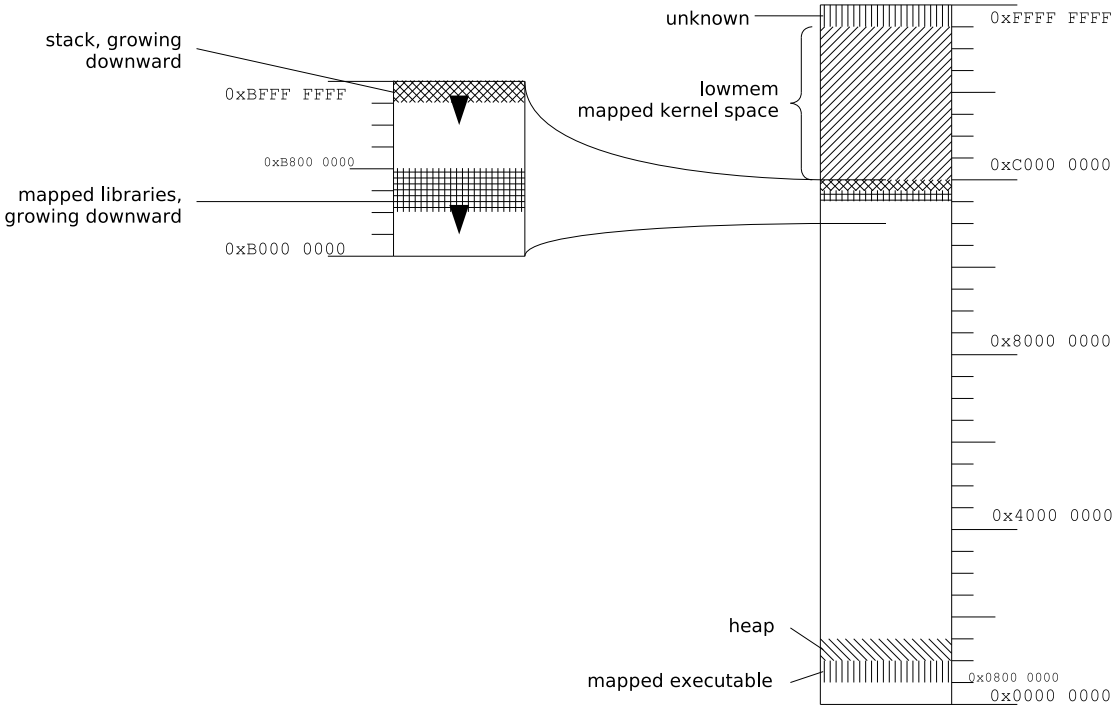

Figure 2: Layout of the virtual address space of a typical Linux process

- *code and heap* will be starting somewhere around `0x0800 0000`, consecutively following with a minor number of unused frames in between

- *libraries* and *custom mappings* will be mapped below `0xb800 0000`
- the *userspace stack* will be mapped somewhere below `0xc000 0000`, possibly directly starting from `0xbfff ffff`
- starting from `0xc000 0000` up to approx. `0xf800 0000` the "lowmem" (approx. lower physical 900 MiB of physical RAM) will be mapped
- the *kernelspace stack* will be located somewhere in this lowmem
- several so far unidentified pages are mapped after `0xf800 0000`
- all *unused frames* will have 4-byte entries consisting of zeroes (`0x00 00 00 00`)

Stack- and memory randomization techniques like *PaX* randomize the base addresses of these locations within several pages, but the general layout stays the same.

Besides searching pages that show non-zero values around these positions and zero values elsewhere, it is much easier and faster to just check, if the virtual address `0xc000 0000` maps to the physical address `0x0`, because typically the PDE for 4MiB-page no. `0x300` will point to the 4MiB physical page at `0x0`. This test only requires reading the 4byte PDE entry `0x300` and does sort out a vast majority of non-pagedirectory pages.

Furthermore in the combination Linux/IA-32 we only have to search the lower 1GB of RAM for pagetables (see first paragraph of section 2.3.1).

3.3.2 Matching via statistics: NCD (normalized compression distance)

For a detailed introduction to and analysis of the NCD and sample applications, the reader may refer to [LV97] and the below given texts.

The *normalized information distance* (NID), a form of parameter-free similarity distance measurement, can be understood as a measurement for the minimal amount of changes required to change one information into another one. A NID of 1 means that two informations are totally unrelated; a NID of 0 means that they are the same. Due to its relation to the *Kolmogorov complexity* (a measurement for an information's shortest description in a fixed description language) it is incalculable. As an approximation, it is possible to use data compressors instead of the Kolmogorov complexity to measure the size of a minimal representation of information.

The resulting *normalized compression distance* has proven to be useful in a vast area of applications; for instance, it has shown its usefulness during analysis of DNA sequences or languages for relatedness ([CV05, LCL+04]), MIDI music files for relations in style and creator ([CV05]) and attack schemes of viruses and worms ([Weh05]).

As the NCD is only an approximation of the NID based on compressors, its resulting "normalized" value can be slightly larger that 1.0 and will never reach 0.

`liblinear` uses the NCD to measure the distance between a known true pagedirectory and a page of unknown data to determine whether this page could be a pagedirectory. The NCD has been chosen, because it is a *parameter-free* measurement, i.e. it does not depend on specific, known structures of the data in question. As different architectures will have significantly different address translation schemes, even depending on the operating systems used, this choice should be adequate. The `complearn`-toolkit [8] provides a suite of functions for generating NCD distance matrices between information, generating relational

[8] `complearn`-toolkit: http://complearn.org/

trees from these and more. As we only need to compare two pages, this set of functions is far too big and the interface far too complex for this application. Thus, we implemented a very short version of the NCD (`simple_ncd()` in `liblinear/simple_ncd.c`) using BZip2[9] as compressor.

4 Attacking

Subsection 4.1 will discuss passive attacks that only *read* from a physical memory source. An overview over gathering information from a virtual address space is given, including finding processes on a host, obtaining its environment, arguments and the path of the processes binary. After that a deeper attack is introduced that is capable of copying ssh public/private keypairs from a running `ssh-agent` process.

In **subsection 4.2**, we will introduce attacks that change data in the virtual address space. Further details about executables, libraries and processes will be given as an introduction. Then we will show how to find mapped libraries, binaries and the stack of a process and how to inject shellcode into a running process (its stack), then introduce a special shellcode that can be used to obtain an interactive shell over firewire only.

4.1 Information Gathering

4.1.1 Identifying processes

Once an address translation table has been found, it is of interest, what kind of process resides in this virtual address space. For userspace applications on IA-32-Linux there is a simple way to identify a process's *filename*, its *arguments* and even its full set of *environment variables*: This information is often required by a process and thus the kernel will provide it to the process by copying it to the bottom pages of the application's stack[10].

`proc_info()` will seek the stack-bottom, parse it and return ready-for-use environment vectors, command-line vectors and the full path of the binary for a given linear address space. `remote-ps`, located in `attacks/information/`, uses `proc_info()` for each found address space and will print a list of all found processes with its arguments.

4.1.2 Typical places to find secrets

Many applications keep secrets in their memory, some of them even locking them into the main memory[11] to prevent the operating system from swapping them to slower (permanent) media. While in general this is a good idea, as an attacker may reconstruct the data from e.g. thrown-away harddisks, it increases the chance of an attacker that can obtain access to the memory of the system in question, as the secret material will be stored in memory completely and unfragmented

"Secrets" includes, among other information, *authentication data*, *cryptographic key material*, *random data* (e.g. to seed a cryptographic algorithm) and sometimes even *algorithms* (*proprietary software*). Authentication data can be e.g. passwords or private keys for signature algorithms. Cryptographic key material, as the name says, are keys for usage with cryptographic algorithms (like the above signature algorithms). These two will be of main interest in this section.

[9] BZip2: A high-quality data compressor, http://www.bzip.org/

[10] i.e. the stack-pages that are found first when seeking downward from virtual address `0xbfff f000`

[11] e.g. via the *mlock* function

Many applications using a cryptographic infrastructure for communications will keep once loaded passwords or keys in their main memory for successive usage. The operating systems protection model ensures the safety of this information from other processes running on the same system; but by accessing the main memory we do have full access to this material. The only task that remains is to reconstruct the key material and passwords from the memory.

As an example, the following applications are of interest:

- GnuPG and PGP: applications to sign and encrypt arbitrary data with public/private keypairs. They are wide spread for email-encryption and -signing.
- sshd, ssh and ssh-agent: the *secure shell* application is an extended, encrypted version of telnet using strong cryptography, including passwords, skey, x509 certificates, RSA and DSA keys.
- Apache and other SSL-enabled web servers.
- OpenVPN, Cisco-VPN and other VPN-servers and clients
- Instant Messaging Applications, e.g. Psi keeps the authentication information and possibly the GnuPG keypair in memory.
- The computer BIOS password, ATA password or PGP-Wholedisk password: the computer or its drives can be locked with a BIOS password or the harddisk can be encrypted. For a sample attack, see [Boi06].

4.1.3 Example attack: ssh-agent snarfer

To proof that it is easy to obtain secret keys from a process we have written a sample attack to obtain (snarf[12]) *ssh public/private keypairs* from `ssh-agent`s via *firewire*.

When using `ssh` for accessing remote computers it is possible to authenticate via passwords, public/private keypairs and various other methods. The usage of public/private keypairs is wide-spread among people using `ssh` on a regular basis. These keypairs can either be a DSA or a RSA keypair, they are typically created with `ssh-keygen` and stored somewhere in `$HOME/.ssh/`, e.g. `/root/.ssh/id_dsa` and `/root/.ssh/id_dsa.pub`. Keypairs can and should be encrypted with a passphrase to prevent attackers from using them, if they were able to obtain them somehow. Thus to use a keypair it is required to enter this passphrase each time. This can be disturbing during frequent usage, e.g. when using `ssh+svn` or `scp` with remote-tab-completion (`zsh` is capable of this).

For these and other reasons, the `ssh-agent` has been developed. This agent will run in the background; the user can store a keypair into it (once entering the passphrase to unlock the keypair) and successively use the keypair without the requirement to enter the passphrase each time. The keypair can be wiped from memory on demand and also be loaded only for a specified period of time.

During our tests we found that the key is *not* wiped from memory when the time limit is hit. It will be wiped the next time the ssh-agent is queried (via its socket), but the agent is stalled in a *read* system call until this query and thus can not wipe the key. That makes

[12] to snarf: To grab, esp. to grab a large document or file for the purpose of using it with or without the author's permission. // To acquire, with little concern for legal forms or politesse (but not quite by stealing). (source: Jargon Files)

it possible to obtain long overdue keys from `ssh-agents`, although their owners believed them to be safe. A simple timer could have prevented this[13]. But even with such a timer enabled it would be possible to acquire the key during its lifetime.

To obtain a keypair from an agent via firewire, a staged attack is required:

1. Seek the first GiB of physical memory for pagetables.

2. For each pagetable: check with the introduced `proc_info()`, if the found userspace belongs to a `ssh-agent` process. If not, seek next pagetable.

3. Use the obtained environment to resolve the users home directory (`$HOME`) and create a path where keypairs most likely reside in the file system (e.g. "`$HOME/.ssh/`") and seek this string in the heap. This approach will only find keypairs that have been loaded with this key-location.[14] Keypairs loaded from different locations or via a relative path can thus not be found by this search.

4. All loaded keypairs have a corresponding *identity-struct* in an agent (see figure 3). Among other fields, this identity-struct contains a link to a key struct, the above mentioned path/comment-field and the lifetime of the key. Thus to find the identity struct corresponding to a found comment-field, one has to search the address of the comment-field in the heap of the agent.

```
1   typedef struct identity {
2           TAILQ_ENTRY(identity) next;
3           Key *key;
4           char *comment;
5           u_int death;
6           u_int confirm;
7   } Identity;
8
9   typedef struct {
10          int nentries;
11          TAILQ_HEAD(idqueue, identity) idlist;
12  } Idtab;
13
14  /* private key table, one per protocol version */
15  Idtab idtable[3];
```

Figure 3: `openssh/ssh-agent.c`: Identity structure and `idtable`

5. Once the *key-struct* (figure 4), that is linked to by the identity-struct, has been found, one can determine whether the found key is a RSA or a DSA key. The key-struct contains a type-field and two pointers to either the RSA or the DSA key. These referenced structures are the *OpenSSL*[15]-structures `RSA` and `DSA`.

6. For both RSA and DSA structures (figure 5 and 6), all important fields need to be recovered to obtain valid keypairs. [Sch96, MvOV01] give an overview of both cryptographic algorithms, [VMC02] introduces OpenSSL concepts and implementation details. OpenSSL's arbitrary precision integer implementation is the `BIGNUM`-struct

[13] We hereby strongly encourage the developers to implement such a timer!

[14] Actually this field is the key's comment-field that is mostly unused and overwritten with the filename of the key. Keypairs that are used with SSH protocol version 2 (virtually all) do not have a comment-field; during loading, the comment-field is always initialized with the keys pathname.

[15] OpenSSL (http://openssl.org) is a free open-source implementation of the secure socket layer protocol also providing a general purpose cryptography library (libcrypto).

```
enum types {
        KEY_RSA1,
        KEY_RSA,
        KEY_DSA,
        KEY_UNSPEC
};

struct Key {
        int       type;
        int       flags;
        RSA      *rsa;
        DSA      *dsa;
};
```

Figure 4: `openssh/key.h`: Key-structure

(often abbreviated "BN"). It consists of a variable-length array of bit-vectors forming the value and a length-field defining the length of this array (see figure 7). As RSA and DSA both operate on finite fields, both are implemented with `BIGNUM`s. Therefore, the RSA and DSA structures contain several `BIGNUM`s that need to be recovered to obtain a valid copy of the keypair.

```
/* Declared already in ossl_typ.h */
/* typedef struct rsa_st RSA; */
struct rsa_st {
        int pad;
        long version;
        const RSA_METHOD *meth;
        ENGINE *engine;
        BIGNUM *n;
        BIGNUM *e;
        BIGNUM *d;
        BIGNUM *p;
        BIGNUM *q;
        BIGNUM *dmp1;
        BIGNUM *dmq1;
        BIGNUM *iqmp;
        int flags;
};
```

Figure 5: `openssl/crypto/rsa/rsa.h`: RSA structure (stripped down)

7. Some validity tests may be done to verify that the acquired `BIGNUM`s fulfill algorithm-specific properties[16] and thus form a valid keypair.

8. Attach the obtained `BIGNUM`s back into valid RSA or DSA structures and save these keys to a file using openssl-functions.

As seen, the search algorithm (2,3,4) has its downsides but works astonishingly well[17]. A much better algorithm to find the *identity*-structs is to use the ELF-headers of the mapped executable to resolve the symbol of the *idtable*[18]. This approach is straightforward, hits *all* keys and should work almost always. The downside of this approach is that most distributions distribute programs with their symbols stripped (due to size and security reasons); this invalidates the symbol-resolution-approach, as this stripping also removes any information of the *idtable* symbol.

[16] During development it was useful to test obtained keys for some properties that are required. These tests are implemented in `attacks/information/sshkey-sanitychecks.c`

[17] To create a better algorithm remains as an exercise to the interested reader.

[18] The *idtable* is a structure referencing all keys that are loaded into the agent.

```
/* Already defined in ossl_typ.h */
/* typedef struct dsa_st DSA; */
struct dsa_st {
        int pad;
        long version;
        int write_params;
        BIGNUM *p;
        BIGNUM *q;       /* == 20 */
        BIGNUM *g;
        BIGNUM *pub_key;  /* y public key */
        BIGNUM *priv_key; /* x private key */
        int flags;
        const DSA_METHOD *meth;
        ENGINE *engine;
};
```

Figure 6: `openssl/crypto/dsa/dsa.h`: DSA structure (stripped down)

```
/* Already declared in ossl_typ.h */
/* typedef struct bignum_st BIGNUM; */
struct bignum_st {
        BN_ULONG *d;     /* Pointer to an array of 'BN_BITS2' bit chunks. */
        int top;         /* Index of last used d +1. */
        /* The next are internal book keeping for bn_expand. */
        int dmax;        /* Size of the d array. */
        int neg;         /* one if the number is negative */
        int flags;
};
```

Figure 7: `openssl/crypto/bn/bn.h`: BIGNUM structure (stripped down)

Once *one identity*-struct is found, *all* structs of the same key-type (RSA or DSA) could be found by walking the list this key is linked into.

As stated above, the keypairs reside decrypted in the memory of the agent (even if overtime) and thus, when snarfed and stored to a file, can be immediately used by the command `ssh -i keyfile user@host`[19]. Such an attack will not take much longer than searching the first 1 GiB of physical RAM for pagedirectories, that is *typically no more than 15 seconds*. If an attack fails but an agent was found, it would be possible to just dump the heap of the agent and stage a more thorough attack at a later time. Once the heap is dumped, all required data is obtained. A similar attack via *ptrace* should be possible as well.

The reader may refer to `attacks/information/snarf-sshkey.c` in the corresponding tarball for the source-code of the attack. Please keep in mind that this attack will only find keys loaded with the absolute path `$HOME/.ssh/`.

4.1.4 Matching and statistics to find secret keys

[SvS99] introduces some schemes to find secret keys in random data and some countermeasures. It takes a special look at finding private RSA keys if their corresponding public keys are known and finding keys by searching high-entropy regions. Though we encourage the reader to read this interesting paper, the circumstances are most likely very different now: cheap and small storage media like flash-memory and small harddisks have increased portable storage to a huge size, equal or larger than the memory a computer system typically has. Thus, an attacker can just dump the full memory or a subset of it (like the virtual address space of a single process) that is promising to contain a secret. A tho-

[19] Only by stealing a key, an attacker will not know, which hosts can be accessed with a retrieved key.

rough attack can then be staged later. Still, searching private keys with the introduced methods can be very helpful, if reconstruction of the used data-structures is impossible or more expensive. Furthermore, when trying to obtain a private key, often enough the corresponding public key is unknown. This invalidates the approaches introduced to find RSA secret keys that require the public key.

4.2 Userspace Modifications

Each executable object, including libraries and executables, can be separated into code and data, where the code should be read-only during execution. The data can further be separated into read-only data, read-write data and uninitialized global variables (local variables will be allocated from the stack, runtime-allocated memory is allocated from the heap). Thus an executable object may be split into four regions: *code*, *rodata* (constants), *rwdata* and *dynamic data* (rwdata may also be implemented by copying the initial data from rodata to dynamic data); stack and heap are process-specific.

As many different processes may use the same executable objects, it would be a waste of memory if the operating system created a copy of the object for each reference to it.

Code and rodata may not be written to by a process, thus the operating system can share these two regions among processes that are using the same objects (binaries or libraries). Thus, once an operating system *ensures* that a process can not write to code-regions and read-only data-regions or introduces a copy-on-write mechanism, it can map these once loaded regions into multiple virtual address spaces. This enforcement is done in hardware, on IA-32 by setting a flag in the page directory or a page table referencing the specific physical memory pages containing the region.

Newer CPUs provide *page-level no-execute enforcements* (AMD's NX *No eXecute bit*, Intel's EXB *EXecute disable Bit*); equal segment-level enforcements exist for years but never have been used in mainstream as Linux and many other operating-systems use a flat memory-model (with only one big segment spanning the full virtual address space). Once these page-level enforcements are used in systems, attacks that inject code into data-regions or the stack are rendered completely useless. However with access to the data-structures (page directory and page tables) containing the information, which pages are executable and which not, one can change this information before injecting code e.g. into an applications stack.

On Linux, programs and libraries are in *Executable and Linker Format* (ELF). This format is described in the manpage `elf(5)`. When a binary is mapped into a process's memory, it is mapped including the full ELF header containing all information that is required to link all references between different objects; ELFs are always mapped at page-bounds. Due to this, all mapped ELFs (that includes executables and libraries) can be found by scanning all pages for the ELF Magic (0x7f E L F) at offset 0 in the page. Libraries, executables and other ELF objects can be distinguished by evaluating the `e_type` field of the ELF header.

4.2.1 Overwriting executable or library code

When code of executables or libraries is changed, all programs using these ELF objects are influenced at the same time. An attacker thus has the ability, but also the burden, to possibly infect several processes at the same time. Such an attack has to be carefully prepared and conducted, as each system may have a different version of a binary and overwriting the wrong parts of an ELF or writing the wrong code may result in an almost

immediate crash of all processes using the ELF. Though this is an interesting approach, there are easier ways to inject code into a *single* process (see 4.2.2).

Such an attack could be conducted by searching a single virtual address space for the glibc and then parsing the ELF-headers and searching for the entry-point of the `printf`-function (or some other, less frequently used function). Then a piece of code could be injected into the unused fragment of the last page of the libc-mapping. It is important to inject the shellcode into the mapping, as all processes that will be affected have to be able to reach the shellcode. The intention is to overwrite the `printf` functions code with a *relative*[20] jump to this injected code. But as we need to overwrite some instructions inside the function, we need to parse the functions code to separate each instruction[21], so that after our code is executed, it executes a copy of the overwritten instructions and jumps to a fully intact instruction right after the injected jump. After this has been done, the jump can be injected into the functions code. This last write has to be as atomic as possible, as a process may just be executing these bytes and thus get astray. On IA-32, entry-points of functions are most likely aligned to 32-bit addresses[22]. Firewire also provides an interface to write 32-bit aligned 32-bit values ("quads") atomically. Unfortunately, a *relative short* jump (2 bytes) can only jump within ±256 bytes from the jump itself and *relative long* and *absolute* jumps are 5 bytes wide (1 byte command + 4 bytes address). A short jump is most likely incapable of reaching the last page of the ELF and writing a long relative jump is not atomic.

A lot of interesting methods to inject code into a running process have been developed; e.g. [Ano02] gives an introduction into using *ptrace*, including injecting whole shared objects using the runtime linker *libdl*. The usability of this approach has not yet been analysed.

4.2.2 Overwriting the stack and return addresses

Besides stealing SSH-keys, we have put most of our efforts into injecting code into the stack and overwriting return-addresses on the stack to point to the injected code. This modification of the classic *stack overflow* method has some advantages over the previous approach:

- Each process, even each thread, has its own stack. Thus only a single thread will be affected by the attack.
- If the attack fails and the thread dies, only a *single* thread will fail on the target system, not e.g. *all* processes using the glibc or *all* instances of */bin/sh*.
- The process can read and write to the stack as well, thus we can communicate with the injected shell-code in a rather easy way (see 4.2.3).
- During the attack we do not need to modify parts of the code of the target-process, reducing the risk of an astray process. The final part consists of overwriting 4 byte wide return-addresses on the stack and this can almost always be done automatically.

[20] An ELF may be mapped at different locations in different processes, if it is "PIC" or "PIE" (Position Independent Code/Executable) and the kernel supports this. Thus, unless the ELF is only mapped in one process or the overwritten function is only used in one process, the jumps target has to be addressed relative to the current position.

[21] on IA-32, different machine instructions can have different length

[22] due to optimizations by the compiler

The attack consists of the following steps:

1. Search a free location in the stack-pages. If the shellcode is small, we can use the zero-padded area of the pages containing the environment and argument vectors (see section 4.1.1). If the shellcode does not fit into this area, we could try to just overwrite these vectors. Most programs will parse environment and arguments only once during startup of the program, thus this should be safe. On the other hand, these vectors are also evaluates when a process accesses a processes *procfs* entries `/proc/$PID/environ` and `/proc/$PID/cmdline`. Thus if these are overwritten with random data, it is possible to see the difference by querying these procfs-entries or using `ps`[23].

2. Scan the stack for stack-frames and for each found: overwrite the return address. This can be simplified into: overwrite all 32-bit aligned 32-bit values that contain a value that might be a pointer into the main code area (`0x08** ****`, see figure 2) with a pointer to the injected code. A more aggressive approach might also overwrite return-values pointing into the library section (`0xB7** ****`).

Once the attacked process leaves a stack frame with an overwritten return value, it will jump to the injected code and execute it.

An implementation of this attack, including some sample shellcodes like a bindshell and a simple `printf` can be found in `attacks/userspace/inject-code.c` and `attacks/userspace/shellcodes/`.

4.2.3 Direct communication with shellcode via DMA

The royal league of shellcode-writing is a shellcode that spawns an interactive shell (thus the name "shellcode") and yet is as invisible on the target system as possible. An interactive shell has to communicate with its user, so typically a network-based bindshell is used for this attack. The downside is the visibility of the communication on the network-layer: an administrator can easily spot the network connection by either sniffing on the network or by asking the system what kind of sockets and files a process is using[24]. A network intrusion system (NIS) can easily spot bindshell connections in an automated way or firewalls could be configured in a way that a network connection is impossible. When using e.g. firewire to attack a host it is even possible that there is no network connection between the attacker and the victim at all.

The bindshell approach has obvious disadvantages. Thus we will use the same attack vector that has been used to inject the "beachhead": physical memory access.

The overall mechanism is introduced in figure 8. The injected shellcode will fork a shell and communicate with stdin/stdout of the shell via two pipes. The shellcode then creates a second thread, then having one thread for each direction of *master to shell* and *shell to master*. If the master (attacker) wishes to send a command to the shell, it writes the command string into the *FromMaster* ring-buffer via DMA. Once the ReaderThread sees that the ring-buffer is not empty, it reads the data inside the buffer and writes it into the pipe to the shell. The WriterThread will read data coming from the shell from the pipe and then write it into the *ToMaster* ring-buffer, so the master can read it via DMA.

[23] `ps` relies on these procfs entries
[24] e.g. by using `lsof -i` (LiSt Open Files)

The shellcode of the beachhead (`attacks/userspace/shellcodes/dmashellcode.s`) is introduced in figure 9 in pseudo-code. The program to inject the shellcode and communicate with it is `attacks/userspace/dmashell.c`.

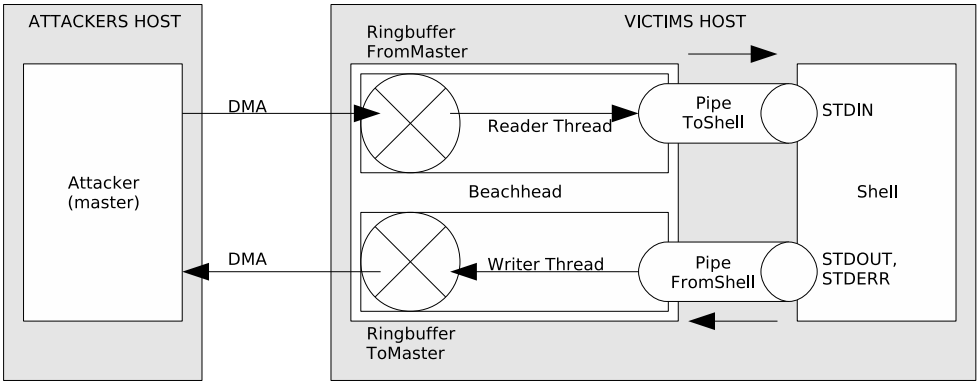

Figure 8: Functionality of the beachhead

5 Future prospects

5.1 Kernelspace Modifications

The Linux character device `/dev/kmem` gives a process read and write access to its virtual address space, including the kernel-space area. `/dev/kmem` has not only been used for its legitimate purpose (i.e. debugging the kernel) but also to inject code into the running kernel and install kernel-based rootkits. [sd01] describes an attack using `/dev/kmem` to inject a new syscall-handler (a regularly used rootkit technique).

5.1.1 Emulating `/dev/kmem`

As `/dev/kmem` gives a process access to its virtual address space without any restrictions, it is yet emulated by `liblinear`, as `liblinear` provides the same functionality, once it has been loaded with a pagetable of a random process. A pagetable of any process will include the fully mapped kernel space ("lowmem", see section 2.3.1 and section 3.3.1).

On the other hand, some new problems come up, if the attacker can not access CPU-registers like the IDT[25], but wishes to know the location of the system call table. If it is necessary to resolve kernel symbols and the to-be-attacked kernel is LKM[26] capable, an attacker could inject a special shellcode to resolve all required symbols via `get_kernel_syms` or inject a LKM and let the kernel do the job. If however an active approach is impossible or the kernel is not LKM capable, statistic approaches might be necessary.

5.2 Bootstrapping custom operating systems

As an attacker has complete access to a systems memory, it is possible to take over the system completely, reset it to a known state and boot a custom operating system on it (and

[25] Interrupt Descriptor Table
[26] Loadable Kernel Module

```
1                   // ringbuffer for data from beachhead to master
2                   RingBuffer      ToMaster;
3                   // ringbuffer for data from master to beachhead
4                   RingBuffer      FromMaster;
5
6                   // pipe for data from beachhead to shell
7                   int             ToShell[2];
8                   // pipe for data from shell to beachhead
9                   int             FromShell[2];
10                  // flag, if pipes are still valid (volatile because two threads access it)
11                  volatile bool   pipes_ok = 1;
12
13                  int             child_pid;
14                  // set by master if child should be killed
15                  volatile bool   terminate_child = 0;
16                  // set by beachhead to signal death of child
17                  bool            child_is_dead = 0;
18                  // set by master to acknowledge death of child
19                  volatile bool   child_is_dead_ACK = 0;
20
21                  start:
22                          pipe(ToShell[]);
23                          pipe(FromShell[]);
24                          switch fork() {
25                                  case child:
26                                          dup2(ToShell[0], stdin);
27                                          dup2(FromShell[1], stdout);
28                                          dup2(FromShell[1], stderr);
29                                          execve("/bin/sh");
30                                          // on failure: exit.
31                                          exit();
32                                          ;;
33                                  case parent:
34                                          child_pid = returnvalue of fork;
35                                          close(ToShell[0]);
36                                          close(FromShell[1]);
37                                          // create a second thread "WriterThread"
38                                          clone(WriterThread);
39                                          // current thread becomes "ReaderThread"
40                                          goto ReaderThread;
41                                          ;;
42                          }
43
44
45                  ReaderThread:
46                          // the ReaderThread will read from the master and relay to the shell
47                          while( pipes_ok ) {
48                                  if(terminate_child) {
49                                          // attacker requested shell to be terminated
50                                          kill(child_pid, SIGKILL);
51                                          pipes_ok = 0;
52                                          // ReaderThread terminates, WriterThread will do cleanup
53                                          exit();
54                                  }
55                                  if( isEmpty(FromMaster) ) {
56                                          sleep(0.001 seconds);
57                                  } else {
58                                          if(1 != write(ToShell[1], FromMaster.buffer
59                                                  +FromMaster.currentLocation, 1 byte)) {
60                                                  // can not communicate with child
61                                                  pipes_ok = 0;
62                                          } else
63                                                  FromMaster.currentLocation += 1;
64                                  }
65                          }
66                          exit();
67
68                  WriterThread:
69                          // the WriterThread will read from the shell and relay to the master
70                          while( pipes_ok ) {
71                                  if( isFull(ToMaster) ) {
72                                          sleep(0.001 seconds);
73                                  } else {
74                                          if(1 != read(FromShell[0], ToMaster.buffer
75                                                  +ToMaster.currentLocation, 1 byte)) {
76                                                  // can not communicate with child
77                                                  pipes_ok = 0;
78                                          } else
79                                                  ToMaster.currentLocation += 1;
80                                  }
81                          }
82                          // pipes are closed. tell the master, but wait at most 2 seconds
83                          while(!chils_is_dead_ACK && child_is_dead <= 2) {
84                                  child_is_dead++;
85                                  sleep(1 second);
86                          }
87                          exit();
```

Figure 9: Beachhead shellcode (`dmashellcode.s`), pseudocode

e.g. use firewire storage as the root-device for the new system). A special bootloader would be required to do this and it might also be necessary to reset the system and attached hardware in a special way, depending on the operating system running on it before the approach.

6 Conclusion

Though some problems remain, it has been shown that firewire and other DMA technology are a mature attack vector having a serious impact on a systems security. DMA interfaces should always be sealed or disabled if untrusted persons can access them; this particularly includes laptops, as more and more of them are equipped with a tiny firewire port. Security "solutions" that deny DMA for some devices and allow DMA for others should be tested very carefully, as these schemes may be fooled by pretending to be a different, "trusted" device (see [Boi06]).

Though most of the tools introduced are designed to attack a system, `libphysical` and `liblinear` can also be used for forensic purposes to analyse memory dumps (with the filedescriptor backend). The statement "There is little experience in reconstructing logical/virtual memory from physical memory dumps" from [BDK05] is no longer true: `liblinear` can be used to access virtual address spaces of each process (independent of the operating system), e.g. IDETECT (by Mariusz Burdach, [Bur06]) can be used to analyse kernel data structures to obtain other information.

7 Acknowledgements

I would like to thank Maximillian Dornseif, Christian N. Klein and Michael Becher for the initial idea and research ([BDK05]), the latter also for personal conversations; the Aachen University of Technology[27] and Lexi Pimenidis for giving me the chance to investigate the idea and write this paper; Timo Boettcher and Alexander Neumann for ideas and comments; Swantje Staar for her support concerning the English language and the Chaos Computer Club Cologne[28] and its members for their support in general concerning projects not limited to, but including this one.

[27] RWTH Aachen, http://www.rwth-aachen.de
[28] Chaos Computer Club Cologne, http://koeln.ccc.de

List of Figures

References

[And99] Don Anderson. *FireWire System Architecture - IEEE 1394*. Addison Wesley, February 1999. ISBN 0201485354.

[Ano02] Anonymous. Runtime process infection. *Phrack, Vol. 0x0b, Issue 0x3b, Phile 0x08*, 2002. http://www.phrack.org/archives/59/p59-0x08.txt.

[BDK05] Michael Becher, Maximillian Dornseif, and Christian N. Klein. Firewire - all your memory are belong to us, 2005. http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf.

[Boi06] Adam Boileau. Ruxcon 2006: Hit by a bus: Physical access attacks with firewire, 2006. http://security-assessment.com/.

[Bur06] Mariusz Burdach. Finding digital evidence in physical memory, 2006. http://forensic.seccure.net/pdf/mburdach_physical_memory_forensics_bh06.pdf and http://forensic.seccure.net/.

[CV05] Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE transactions on information theory, vol. 51*, 2005. http://www.cwi.nl/ paulv/papers/cluster.pdf.

[Int06a] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, 2006. http://developer.intel.com/.

[Int06b] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*, 2006. http://developer.intel.com/.

[Int06c] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, 2006. http://developer.intel.com/.

[Int06d] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2006. http://developer.intel.com/.

[Int06e] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, 2006. http://developer.intel.com/.

[LCL+04] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M. B. Vitányi. The similarity metric. *IEEE transactions on information theory*, August 2004. http://arxiv.org/pdf/cs.CR/0111054.

[LV97] Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications (2nd Edition)*. Springer Verlag, 1997. ISBN 0387948686.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, August 2001. available at http://www.cacr.math.uwaterloo.ca/hac/.

[Pro00] Promoters of the 1394 Open HCI. *1394 Open Host Controller Interface Specification*, January 2000.

[S+02] Otto Spaniol et al. *Systemprogrammierung, Skript zur Vorlesung an der RWTH Aachen*. Wissenschaftsverlag Mainz; Aachener Beitraege zur Informatik (ABI), 2002. ISBN 3-86073-470-9.

[Sch96] Bruce Schneier. *Applied Cryptography (Second Edition)*. John Wiley & Sons, Inc, 1996. ISBN 0-471-11709-9.

[sd01] sd and devik. Linux on-the-fly kernel patching without lkm. *Phrack, Vol. 0x0b, Issue 0x3a, Phile 0x07*, December 2001. http://www.phrack.org/archives/58/p58-0x07.txt.

[SvS99] Adi Shamir and Nicko van Someren. Playing "hide and seek" with stored keys. *Lecture Notes in Computer Science*, 1648:118–124, 1999.

[VMC02] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL*. O'Reilly, 2002. ISBN 0-596-00270-X.

[Weh05] Stephanie Wehner. Analyzing worms and network traffic using compression, April 2005. http://arxiv.org/pdf/cs.CR/0504045.