

DIMVA 2007, Lucerne
Targeting Physically Addressable Memory

David Rasmus Piegdon

<david.rasmus.piegdon@rwth-aachen.de>

Lexi Pimenidis

<lexi@i4.informatik.rwth-aachen.de>

Lehrstuhl für Informatik IV, RWTH Aachen

<http://www-i4.informatik.rwth-aachen.de>

July 13th 2007



Table of Contents

- 1 Accessing memory
- 2 Virtual address spaces
- 3 Gathering information
- 4 Injecting code
- 5 Prospects, Conclusion



Table of Contents

- 1 Accessing memory
- 2 Virtual address spaces
- 3 Gathering information
- 4 Injecting code
- 5 Prospects, Conclusion



Methods

Many ways to gain access to memory:

- special PCI cards (forensic, remote management cards)
- special PCMCIA cards
- FireWire (IEEE1394) DMA feature
- anything with DMA
- Linux `/dev/mem`
- memory dumps
- Suspend2Disk images
- Virtual machines
- ...



FireWire overview



FireWire a.k.a. iLink a.k.a. IEEE1394

- Hot-pluggable
- Wide-spread (even among laptops)
- Expansion Bus (like PCI or PCMCIA)
- Has DMA (if enabled by driver)
- Most people are not aware of abuse-factor



/dev/mem

- Gives access to physically addressed memory
- Shall be obsoleted in future (X shall use DRI)
- Only gives access to **lower 896MB** RAM (only these are mapped)



Generic problems of DMA attacks

- Swapping
- Multiple accessors at any time
- Caching



Countermeasures against DMA

Several approaches, both to hide malware from forensics and to protect memory from DMA attacks:

- Keep stealth code/data only in CPU cache (?)
- Redirect CPU via MTTR/IORR registers [9]
- Redirect DMA via NorthBridge (IOMMU) [9]



One interface to access any source

- One generic interface: `libphysical`
- Backends for any source
- Implemented so far:
 - Filedescriptor (`/dev/mem`, memory dumps)
 - FireWire



Table of Contents

- 1 Accessing memory
- 2 Virtual address spaces
- 3 Gathering information
- 4 Injecting code
- 5 Prospects, Conclusion



What now?

- Once we got access. . . we can see a bunch of random memory
- **How does OS manage memory?**
 - Parse kernel structures (depends on Arch,OS+Version)
 - Parse virtual address spaces (depends on Arch)

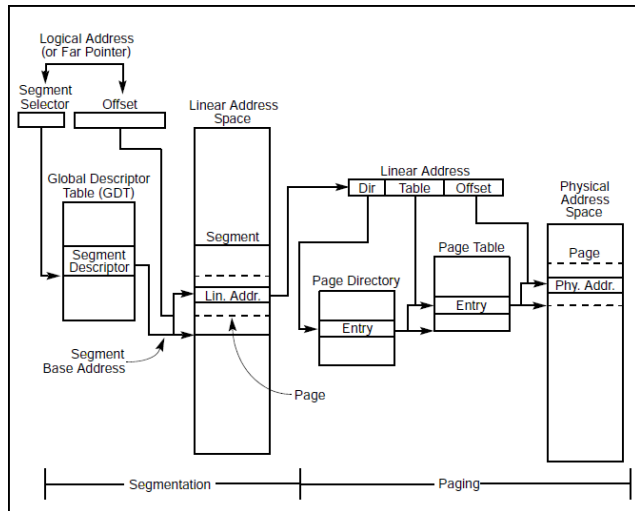


IA-32

- IA-32 provides **Segmentation** (required) and **Paging** (optional)
- Linux only uses paging → all segments flat
- Hardware needs to know how to translate:
 - Global Descriptor Table (GDT) (segmentation)
 - Local Descriptor Table (LDT) (segmentation)
 - Page Directory (PD)
 - Page Tables (PT) (referenced by PD)



IA-32 virtual (“logical”) address translation



(from [6])



Finding ATTs

Address Translation Tables (including PDs)...

- depend on **architecture**
- depend on **operating system**
- may have recognisable patterns

→ create simple signature for (arch, OS). so far:

- (i386, Linux 2.4 and 2.6)
- (i386, Windows XP)

→ Finding an ATT:

- Check for simple pattern AND do statistical test (NCD)



Normalized Compression Distance

- Normalized Information Distance:
 - Minimal amount of changes required between two information
 - Uses Kolmogorov Complexity (KC) (size of minimal representation of information)
 - Incalculable
- KC can be approximated by compressor
 - Normalized Compression Distance:
 - Calculable
 - Very versatile (parameter-free)
 - e.g. create relational trees of gene-sequences [4]
 - or *Analyzing Worms and Network Traffic using Compression*, Stephanie Wehner, 2006



liblinear

Translation by hand:

- Implementation in software in `liblinear`. So far:
 - IA-32 Protected Mode, without PAE36
(Linux with $\leq 4\text{GB}$ RAM)
- Well-defined algorithm for architecture, e.g. for IA-32: [6]
- emulates `/dev/kmem` on the fly
- → also for forensics



Table of Contents

- 1 Accessing memory
- 2 Virtual address spaces
- 3 Gathering information
- 4 Injecting code
- 5 Prospects, Conclusion



So far

- We can **access physical memory sources** in a generic way (`libphysical`)
- We can find and **access virtual address spaces** of processes (`liblinear`)

Now we want to **identify** processes we found.



```
#include <stdio.h>

int main(int argc, char**argv)
{
    printf("my name is %s\n", argv[0]);
    return 0;
}
```

- `argv`, `envv` are on the stack (first mapped pages below page `0xc0000`)
- NUL-separated vector with
 - Path of binary
 - Environment
 - Arguments



Identifying Processes

```
# OLDPWD=/home/lostrace PWD=/home/lostrace/documents/rwth/SEAT \
  /attacks/userspace SHLVL=1 _=./victim \
  ./victim --arg=foo bar --baz
```

0xbfc5ff70	00 00 00 00	2e 2f 76 69/vi	ARGV[]:
0xbfc5ff78	63 74 69 6d	00 2d 2d 61	ctim.--a	[0] = bfc5ff74
0xbfc5ff80	72 67 3d 66	6f 6f 00 62	rg=foo.b	[1] = bfc5ff7d
0xbfc5ff88	61 72 00 2d	2d 62 61 7a	ar.--baz	[2] = bfc5ff87
0xbfc5ff90	00 4f 4c 44	50 57 44 3d	.OLDPWD=	[3] = bfc5ff8b
0xbfc5ff98	2f 68 6f 6d	65 2f 6c 6f	/home/lo	[4] = NULL
0xbfc5ffa0	73 74 72 61	63 65 00 50	strace.P	
0xbfc5ffa8	57 44 3d 2f	68 6f 6d 65	WD=/home	
0xbfc5ffb0	2f 6c 6f 73	74 72 61 63	/lostrac	
0xbfc5ffb8	65 2f 64 6f	63 75 6d 65	e/docume	
0xbfc5ffc0	6e 74 73 2f	72 77 74 68	nts/rwth	
0xbfc5ffc8	2f 53 45 41	54 2f 61 74	/SEAT/at	
0xbfc5ffd0	74 61 63 6b	73 2f 75 73	tacks/us	
0xbfc5ffd8	65 72 73 70	61 63 65 00	erspace.	
0xbfc5ffe0	53 48 4c 56	4c 3d 31 00	SHLVL=1.	
0xbfc5ffe8	5f 3d 2e 2f	76 69 63 74	_=./vict	
0xbfc5fff0	69 6d 00 2e	2f 76 69 63	im../vic	
0xbfc5fff8	74 69 6d 00	00 00 00 00	tim.....	



Finding Specific Processes

- 1 Find all virtual address spaces
- 2 For each: look if binary matches searched binary
 - /usr/lib/mozilla-firefox/firefox-bin
 - /usr/bin/gpg
 - /usr/bin/ssh-agent
- 3 If matches, steal a cookie or... a ssh-private key



Finding Specific Processes

- 1 Find all virtual address spaces
- 2 For each: look if binary matches searched binary
 - /usr/lib/mozilla-firefox/firefox-bin
 - /usr/bin/gpg
 - /usr/bin/ssh-agent
- 3 If matches, steal a cookie or... a ssh-private key



Stealing SSH private keys

Let's get dangerous!

Steal SSH private key from `ssh-agent`:

- agent keeps key decrypted, locked in memory
- has timeout-function to wipe keys from memory
- stalled in `read()`-syscall on socket
- **no timer** to check for timeout (Fixed by now)
- checks timer only on query (Fixed by now)



```

typedef struct identity {
    Key *key;
    char *comment;
        (key filename: "$HOME/.ssh/id_rsa")
    u_int death;
} Identity;

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

```



Resume

- So far: only **read** memory.
- Works with memory dumps
- No time to prepare an attack?
- → Just dump memory and do it later



Table of Contents

- 1 Accessing memory
- 2 Virtual address spaces
- 3 Gathering information
- 4 Injecting code**
- 5 Prospects, Conclusion



Inject where?

- Cannot allocate memory for code
- Cannot overflow a buffer (no I/O with process)
- Need to **overwrite** code, data or stack
- Code: **Shared** objects/executable
- Data: can not be easily identified; data may be mapped into multiple processes



Inject into stack

- Stack is easy to find
- One stack per thread
- Inject into zero-padded pages containing ENV and ARG.
- Possibly overwrite these. If so, visible:
 - `/proc/$PID/environ`
 - `/proc/$PID/cmdline`
- `execute`: manipulate stack frames (checksums? haha!)



Rootshell?

- Royal leage of code-injection: [interactive \(root-\)shell](#)
- Inject bindshell
- Network required
 - Easily found
- Inject shellcode and communicate via IEEE1394
- Big, complex payload (IEEE1394 handling)
 - Attack via IEEE1394?
- Inject Syscall-Proxy
- Victim, self need to be same Arch, OS, API/ABI
 - I attacked IA-32 from PPC. . .

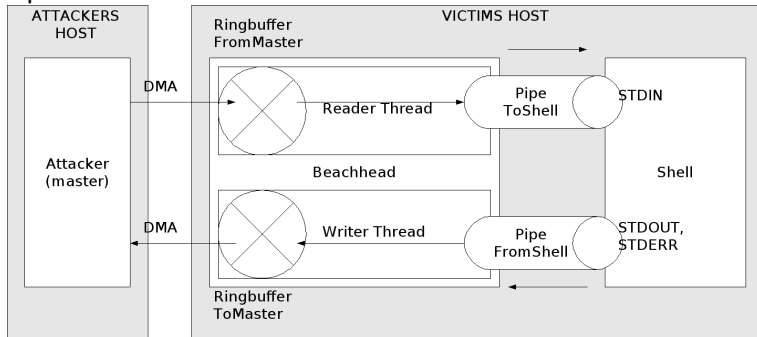


DMA-Shell

- Only thing that is for sure: DMA
- Communication via DMA



Special “Beachhead” Shellcode:



- Payload small (536 Bytes, yet big for shellcode)
- Independent of attackers arch, OS
- Only DMA required



Table of Contents

- 1 Accessing memory
- 2 Virtual address spaces
- 3 Gathering information
- 4 Injecting code
- 5 Prospects, Conclusion



Prospects

- Kernelspace Modifications:
 - Shellcode that injects LKM?
 - Live kernel patching?
- Bootstrapping custom operating systems



Conclusion

- DMA attacks are mature
- Access to memory → security=0
- Keep your firewire-ports secured
- Some of the tools (`libphysical`, `liblinear`) can also be used for forensics



Questions?

Thank you for your attention!

Paper/Tools have been released at
<http://david.piegdon.de/products.html>

Thanks to Maximillian Dornseif, Christian N. Klein, Michael Becher, Timo Boettcher, Alexander Neumann, Swantje Staar and the Chaos Computer Club Cologne



References (FireWire, DMA Attacks)

- 1 *Michael Becher, Maximillian Dornseif, and Christian N. Klein.* [Firewire - all your memory are belong to us](#). 2005.
- 2 *Adam Boileau.* Hit by a bus: Physical access attacks with firewire. Ruxcon 2006.
- 3 *Mariusz Burdach.* Finding digital evidence in physical memory. 2006.



References

- 4 *Rudi Cilibrasi and Paul M. B. Vitányi*. Clustering by compression. IEEE transactions on information theory, vol. 51, 2005.
- 5 *Otto Spaniol et al.* Systemprogrammierung, Skript zur Vorlesung an der RWTH Aachen. Wissenschaftsverlag Mainz; Aachener Beiträe zur Informatik (ABI), 2002. ISBN 3-86073-470-9.
- 6 *Intel Corp.* Intel 64 and IA-32 Architectures Software Developers Manual.



References

- 7 *Bruce Schneier*. Applied Cryptography (Second Edition). John Wiley & Sons, Inc, 1996. ISBN 0-471-11709-9.
- 8 *John Viega and Matt Messier and Pravir Chandra*. Network Security with OpenSSL. O'Reilly, 2002. ISBN 0-596-00270-X.
- 9 *Joana Rutkowska*. Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case)

