# Betriebssysteme KU Security

**IAIK**
**Graz University of Technology**

November 27, 2014

1. The simple stuff

2. Code injection attacks

3. Cache attacks

1. The simple stuff
2. Code injection attacks
3. Cache attacks

# Multi user environments

- Usermanagement
- File access rights

→ Extend the existing file system

# DOS attacks

- Denial of service $\rightarrow$ using an unfairly amount of resources to block the whole system

# DOS attacks

- Denial of service $\rightarrow$ using an unfairly amount of resources to block the whole system
- Improve the Scheduler (priorities)
- Scheduler for I/O activity?

# DOS attacks

- Denial of service $\rightarrow$ using an unfairly amount of resources to block the whole system
- Improve the Scheduler (priorities)
- Scheduler for I/O activity?
- Out of memory handling?
- `ulimit` as an operating system service

# Private exec

- Modern browsers have a “private browsing” mode
- Shouldn’t that be a service of the operating system?

# Private exec

- Modern browsers have a “private browsing” mode
- Shouldn’t that be a service of the operating system?
- `priv_exec` $\rightarrow$ execute a program in a sandbox

$\rightarrow$ No accesses to real file system, ...

$\rightarrow$ No interaction with any hardware

$\rightarrow$ Maybe a limited set of syscalls?

1. The simple stuff

2. Code injection attacks

3. Cache attacks

# Code injection attack

Idea: Use buffer overflow to inject binary code

```
int f()
{
  char b[128];
  gets(b);
}
```

How does the stack look now?

# Code injection attack

Stack:

0xfc return address

0xf8 `ebp` from calling function

0xf4 `b[124],b[125],b[126],b[127]`

... ...

# Code injection attack

→ Input string has 128 bytes + attack payload:

```
"\0\1\2\3"          // the ebp i choose
"\xe0\x05\x00\x08" // 080005e0 - execv
"\4\5\6\7"          // argument: char* path
"\8\9\10\11"        // argument: char** argv
```

# Countermeasures against code injection

- NX-bit in page table $\rightarrow$ prevent execution on a page
- Use this bit for stack pages:
  PageFault if `eip` points on a stack page

# Countermeasures against code injection

- NX-bit in page table $\rightarrow$ prevent execution on a page
- Use this bit for stack pages:
  PageFault if `eip` points on a stack page
- Think of the attack we just saw:
  did we execute code on the stack?

# Countermeasures against code injection

- NX-bit in page table $\rightarrow$ prevent execution on a page
- Use this bit for stack pages:
  PageFault if `eip` points on a stack page
- Think of the attack we just saw:
  did we execute code on the stack?

$\rightarrow$ Return-to-libc/Return-Oriented-Programming (ROP) still possible! Any ideas?

# Countermeasures against code injection

- NX-bit in page table $\rightarrow$ prevent execution on a page
- Use this bit for stack pages:
  PageFault if `eip` points on a stack page
- Think of the attack we just saw:
  did we execute code on the stack?

$\rightarrow$ Return-to-libc/Return-Oriented-Programming (ROP) still possible! Any ideas?

- Randomize the position of (shared) library code!

# Countermeasures against code injection

- Stack canaries $\rightarrow$ detect stack corruption
- Kernel stops execution if stack is corrupted
  $\rightarrow$ check during context switch

# Countermeasures against code injection

- Stack canaries $\rightarrow$ detect stack corruption
- Kernel stops execution if stack is corrupted $\rightarrow$ check during context switch
- Who protects the kernel against code injection attacks?

# Syscall-based countermeasures

If injection only changes arguments passed to `__syscall`:

- Randomize syscall numbers
- Randomize syscall argument order
- Blacklist syscalls
- Whitelist syscalls
- ...

# NX-Bit

- Execution Prevention for stack pages... what more?

# NX-Bit

- Execution Prevention for stack pages... what more?
- Binary might prefer non-writeable code pages and non-executable data pages

# NX-Bit

- Execution Prevention for stack pages... what more?
- Binary might prefer non-writeable code pages and non-executable data pages
- Is there any reason to have a page writeable **and** executable? (apart from self-modifying code)

# NX-Bit

- Execution Prevention for stack pages... what more?
- Binary might prefer non-writeable code pages and non-executable data pages
- Is there any reason to have a page writeable **and** executable? (apart from self-modifying code)
- W$\oplus$X policy $\rightarrow$ no page writeable and executable at the same time

# NX-Bit

- Execution Prevention for stack pages... what more?
- Binary might prefer non-writeable code pages and non-executable data pages
- Is there any reason to have a page writeable **and** executable? (apart from self-modifying code)
- W$\oplus$X policy $\rightarrow$ no page writeable and executable at the same time (except if the binary wants self-modifying code)

# Code injection in kernel

- What if you can only write a very small amount of data in the kernel?
- Where to jump?

# Code injection in kernel

- What if you can only write a very small amount of data in the kernel?
- Where to jump?
- We should prevent the kernel from being able to execute userspace code
- Now think of Return-to-libc/ROP...

# Code injection in kernel

- What if you can only write a very small amount of data in the kernel?
- Where to jump?
- We should prevent the kernel from being able to execute userspace code
- Now think of Return-to-libc/ROP...
- What if we can set `ebp`/`esp` to point into userspace?

# Code injection in kernel

- Maybe the kernel should not have userspace data mapped?

# Code injection in kernel

- Maybe the kernel should not have userspace data mapped?

$\rightarrow$ We remove the userspace mapping from kernel

- Linux did that too, but they are still vulnerable... why?

# Code injection in kernel

- Maybe the kernel should not have userspace data mapped?

→ We remove the userspace mapping from kernel

- Linux did that too, but they are still vulnerable... why?

→ Identity mapping breaks everything!

1. The simple stuff

2. Code injection attacks

3. Cache attacks

# Cache timing

- Different cache attacks - we will only talk about a very simple one: Flush+Reload

# Cache timing

- Different cache attacks - we will only talk about a very simple one: Flush+Reload
- Shared libraries $\rightarrow$ different process execute code on identical physical pages

# Cache timing

- Different cache attacks - we will only talk about a very simple one: Flush+Reload
- Shared libraries $\rightarrow$ different process execute code on identical physical pages
- Shared pages $\rightarrow$ shared location in cache

# Cache timing

- Different cache attacks - we will only talk about a very simple one: Flush+Reload
- Shared libraries $\rightarrow$ different process execute code on identical physical pages
- Shared pages $\rightarrow$ shared location in cache
- When we read data from a shared page: Can we sense whether it is in cache or in RAM? How?

# Cache timing

- Different cache attacks - we will only talk about a very simple one: Flush+Reload
- Shared libraries $\rightarrow$ different process execute code on identical physical pages
- Shared pages $\rightarrow$ shared location in cache
- When we read data from a shared page: Can we sense whether it is in cache or in RAM? How?
- Timing: if memory access takes $<$ 50ns $\rightarrow$ Cache, else RAM
- How to measure time?

# Cache timing

- Different cache attacks - we will only talk about a very simple one: Flush+Reload
- Shared libraries $\rightarrow$ different process execute code on identical physical pages
- Shared pages $\rightarrow$ shared location in cache
- When we read data from a shared page: Can we sense whether it is in cache or in RAM? How?
- Timing: if memory access takes $<$ 50ns $\rightarrow$ Cache, else RAM
- How to measure time?$\rightarrow$ `rdtsc`!

# Flush and Reload

- Idea: Access memory in a high frequency to sense accurately when victim executes certain code

# Flush and Reload

- Idea: Access memory in a high frequency to sense accurately when victim executes certain code
- Example: Whether algorithm performed square or multiply $\rightarrow$ retrieve RSA key

# Flush and Reload

- Idea: Access memory in a high frequency to sense accurately when victim executes certain code
- Example: Whether algorithm performed square or multiply $\rightarrow$ retrieve RSA key
- After accessing memory it stays in the cache. How to solve that?

# Flush and Reload

- Idea: Access memory in a high frequency to sense accurately when victim executes certain code
- Example: Whether algorithm performed square or multiply $\rightarrow$ retrieve RSA key
- After accessing memory it stays in the cache. How to solve that?
  - Filling the cache with garbage (we call that evict)

# Flush and Reload

- Idea: Access memory in a high frequency to sense accurately when victim executes certain code
- Example: Whether algorithm performed square or multiply $\rightarrow$ retrieve RSA key
- After accessing memory it stays in the cache. How to solve that?
  - Filling the cache with garbage (we call that evict)
  - x86 has a flush instruction to invalidate a cache line

# Flush and Reload

- Powerful attack

# Flush and Reload

- Powerful attack
- Accurate! (one trace is often enough to retrieve cryptographic key)

# Flush and Reload

- Powerful attack
- Accurate! (one trace is often enough to retrieve cryptographic key)
- Cross-VM in the Cloud (works for instance on Amazon cloud VMs)

# Cache attack countermeasures

- OS could flush cache during context switch

# Cache attack countermeasures

- OS could flush cache during context switch
- Maybe not the whole cache, but only parts of it

# Cache attack countermeasures

- OS could flush cache during context switch
- Maybe not the whole cache, but only parts of it
- User programs could register which parts they want to protect

# EOF

Better ideas? We want to see them!

Have fun programming!