OS Security

Mandatory Access Control

Radboud University Nijmegen, The Netherlands

Winter 2015/2016

Exam date

- The exam is on **Monday, January 18, 12:30–15:30** in Lin 4/5!
- Last exercise class (Q&A): Tomorrow 10:30 in HG00.062.

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this
  - Full virtualization (guest OS in ring 1)

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this
  - Full virtualization (guest OS in ring 1)
  - HW-assisted virtualization (guest OS in ring 0, hypervisor in ring -1)

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this
  - Full virtualization (guest OS in ring 1)
  - HW-assisted virtualization (guest OS in ring 0, hypervisor in ring -1)
  - Paravirtualization: modified guest OS in ring 3, host OS in ring 1, hypervisor in ring 0

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this
  - Full virtualization (guest OS in ring 1)
  - HW-assisted virtualization (guest OS in ring 0, hypervisor in ring -1)
  - Paravirtualization: modified guest OS in ring 3, host OS in ring 1, hypervisor in ring 0
  - Host-based virtualization: hypervisor in ring 3, unmodified guest OS in ring 3

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this
  - Full virtualization (guest OS in ring 1)
  - HW-assisted virtualization (guest OS in ring 0, hypervisor in ring -1)
  - Paravirtualization: modified guest OS in ring 3, host OS in ring 1, hypervisor in ring 0
  - Host-based virtualization: hypervisor in ring 3, unmodified guest OS in ring 3
- Desktop-oriented OS for high security: Qubes
- Idea: User defines security domains, those are separated by virtualization

A short recap

- Important concept to reduce covert channels and possible damage by an attack: compartmentalization
- Very strong implementation: virtualization
- Multiple *virtual machines* running on the same physical hardware
- Different ways to implement this
  - Full virtualization (guest OS in ring 1)
  - HW-assisted virtualization (guest OS in ring 0, hypervisor in ring -1)
  - Paravirtualization: modified guest OS in ring 3, host OS in ring 1, hypervisor in ring 0
  - Host-based virtualization: hypervisor in ring 3, unmodified guest OS in ring 3
- Desktop-oriented OS for high security: Qubes
- Idea: User defines security domains, those are separated by virtualization
- Weaker way to compartmentalize: sandboxing
- Sandboxing limits access to resources
- Attacks typically aim at breaking out of the jail

X “security”

X "security"

- X window system is designed with no security in mind
- Processes running on the same X server have *no isolation*

X "security"

- X window system is designed with no security in mind
- Processes running on the same X server have *no isolation*
- Simple exeriment (See http://theinvisiblethings.blogspot.nl/2011/04/linux-security-circus-on-gui-isolation.html)
  - Run `xinput list`, remember **id** of AT keyboard
  - Run `xinput test` **id**
  - In a different window, enter some text
  - `xinput` will *read all keystrokes*
  - ... even keystrokes of processes by other users
  - This is not a bug or exploit or anything, this is X design!

X "security"

- X window system is designed with no security in mind
- Processes running on the same X server have *no isolation*
- Simple exeriment (See http://theinvisiblethings.blogspot.nl/2011/04/linux-security-circus-on-gui-isolation.html)
  - Run `xinput list`, remember **id** of AT keyboard
  - Run `xinput test` **id**
  - In a different window, enter some text
  - `xinput` will *read all keystrokes*
  - ... even keystrokes of processes by other users
  - This is not a bug or exploit or anything, this is X design!
- No UNIX permissions stop this!
- No SELinux/AppArmor (this lecture) stops this!
- Qubes OS prevents this kind of information flow between different security domains (VMs)

A somewhat longer recap

- Traditional UNIX security uses discretionary access control
- Each user decides about access permissions of his/her files
- Modern attack scenarios:
  - User runs malware, malware sends private data through Internet (confidentiality)
  - User runs malware, malware modifies user's files (integrity)
- DAC cannot prevent this kind of attack
- Compartmentalization can only limit scope of such an attack
- Protecting system-level information flow needs MAC

MAC and LSM

- Linux security traditionally follows the UNIX security model
- Around 2000, various projects worked on MAC (and generally stronger security) for Linux
- Linus Torvalds about inclusion of SELinux: “make it a module”

MAC and LSM

- Linux security traditionally follows the UNIX security model
- Around 2000, various projects worked on MAC (and generally stronger security) for Linux
- Linus Torvalds about inclusion of SELinux: "make it a module"
- Since Kernel 2.6: API for *Linux Security Modules* (LSMs)
- Hooks to module functions when accessing security-critical resources

MAC and LSM

- Linux security traditionally follows the UNIX security model
- Around 2000, various projects worked on MAC (and generally stronger security) for Linux
- Linus Torvalds about inclusion of SELinux: "make it a module"
- Since Kernel 2.6: API for *Linux Security Modules* (LSMs)
- Hooks to module functions when accessing security-critical resources
- An LSM sets function pointers in a data structure called `security_operations`
- Global table of this type called `security_ops` defined in `include/linux/security.h`

Criticism of LSM

LSM is in the mainline kernel and various LSM implemetations exist, however, there is some criticism of the API:

- Small overhead even if no LSM is loaded

Criticism of LSM

LSM is in the mainline kernel and various LSM implemetations exist, however, there is some criticism of the API:

- Small overhead even if no LSM is loaded
- LSM is designed for access control, but can be abused, for example, for bypassing GPL license

Criticism of LSM

LSM is in the mainline kernel and various LSM implemetations exist, however, there is some criticism of the API:

- Small overhead even if no LSM is loaded
- LSM is designed for access control, but can be abused, for example, for bypassing GPL license
- "Because LSM is compiled and enabled in the kernel, its symbols are exported. Thus, every rootkit and backdoor writer will have every hook he ever wanted in the kernel."
  (`https://grsecurity.net/lsm.php`)

Criticism of LSM

LSM is in the mainline kernel and various LSM implemetations exist, however, there is some criticism of the API:

- Small overhead even if no LSM is loaded
- LSM is designed for access control, but can be abused, for example, for bypassing GPL license
- "Because LSM is compiled and enabled in the kernel, its symbols are exported. Thus, every rootkit and backdoor writer will have every hook he ever wanted in the kernel." (https://grsecurity.net/lsm.php)
- LSM provides hooks only for access control
- Systems like grsecurity and RSBAC need more than just access control

Criticism of LSM

LSM is in the mainline kernel and various LSM implemetations exist, however, there is some criticism of the API:

- Small overhead even if no LSM is loaded
- LSM is designed for access control, but can be abused, for example, for bypassing GPL license
- "Because LSM is compiled and enabled in the kernel, its symbols are exported. Thus, every rootkit and backdoor writer will have every hook he ever wanted in the kernel." (https://grsecurity.net/lsm.php)
- LSM provides hooks only for access control
- Systems like grsecurity and RSBAC need more than just access control
- "Stacking" multiple security modules is problematic

Criticism of LSM

LSM is in the mainline kernel and various LSM implemetations exist, however, there is some criticism of the API:

- Small overhead even if no LSM is loaded
- LSM is designed for access control, but can be abused, for example, for bypassing GPL license
- "Because LSM is compiled and enabled in the kernel, its symbols are exported. Thus, every rootkit and backdoor writer will have every hook he ever wanted in the kernel." (https://grsecurity.net/lsm.php)
- LSM provides hooks only for access control
- Systems like grsecurity and RSBAC need more than just access control
- "Stacking" multiple security modules is problematic
- LSM hooks expose kernel internal data structures as parameters

Implementations of LSM

- AppArmor
- Linux Intrusion Detection System (LIDS)
- POSIX capabilitites
- Simplified Mandatory Access Control Kernel (Smack)
- TOMOYO
- Security-Enhanced Linux (SELinux)

SELinux overview

- Originally developed by the NSA
- Used today by, for example, Red Hat Linux, Fedora, CentOS

SELinux overview

- Originally developed by the NSA
- Used today by, for example, Red Hat Linux, Fedora, CentOS
- Provides three kinds of MAC mechanisms:
  1. Type enforcement (TE)
  2. Role-based access control
  3. Multi-level security (MLS)

SELinux overview

- Originally developed by the NSA
- Used today by, for example, Red Hat Linux, Fedora, CentOS
- Provides three kinds of MAC mechanisms:
  1. Type enforcement (TE)
  2. Role-based access control
  3. Multi-level security (MLS)
- All approaches are *additional* to UNIX DAC: first check file permissions, if those allow access additionally check MAC rules.

Type Enforcement

- All subjects and objects have a security context in the format

  `user:role:type`

- Mainly important for the moment: the `type`

Type Enforcement

- All subjects and objects have a security context in the format

  `user:role:type`

- Mainly important for the moment: the `type`
- Obtain security context using classical Linux commands with `-Z`, e.g.,
  - `ps -Z` shows processes with security context
  - `id -Z` shows security context of current user
  - `ls -Z` shows security context of files
  - `netstat -Z` shows secuirty context of network sockets

Type Enforcement

- All subjects and objects have a security context in the format

  `user:role:type`

- Mainly important for the moment: the `type`
- Obtain security context using classical Linux commands with `-Z`, e.g.,
  - `ps -Z` shows processes with security context
  - `id -Z` shows security context of current user
  - `ls -Z` shows security context of files
  - `netstat -Z` shows secuirty context of network sockets
- All access has to be explicitely granted, using allow rules
- Format:

  `allow source_type target_type : object_class permissions;`

Type Enforcement

- All subjects and objects have a security context in the format

  `user:role:type`

- Mainly important for the moment: the `type`
- Obtain security context using classical Linux commands with `-Z`, e.g.,
  - `ps -Z` shows processes with security context
  - `id -Z` shows security context of current user
  - `ls -Z` shows security context of files
  - `netstat -Z` shows secuirty context of network sockets
- All access has to be explicitely granted, using allow rules
- Format:

  `allow source_type target_type : object_class permissions;`

- Example:

  `allow user_t bin_t : file {read execute getattr};`

  "A process with domain type (source type) `user_t` can read, execute, or get attributes for a file object with (target type) of `bin_t`."

Type Enforcement ctd.

- Default assignment of security context:
  - processes get the context of the parent process
  - files get the context of the parent directory

Type Enforcement ctd.

- Default assignment of security context:
  - processes get the context of the parent process
  - files get the context of the parent directory
- Various ways to change this behavior
- Most important, transition rules:
  `type_transition source_type target_type : class new_type;`

Type Enforcement ctd.

- Default assignment of security context:
  - processes get the context of the parent process
  - files get the context of the parent directory
- Various ways to change this behavior
- Most important, transition rules:

```
type_transition source_type target_type : class new_type;
```

- Example:

```
type_transition httpd_t httpd_sys_script_exec_t : \
      process httpd_sys_script_t;
```

"When the httpd daemon running in the domain `httpd_t` executes a program of the type `httpd_sys_script_exec_t`, such as a CGI script, the new process is given the domain of `httpd_sys_script_t`"

Type Enforcement vs. DAC

- SELinux TE can be used to separate security domains
- This is separation on a different layer than virtualization

Type Enforcement vs. DAC

- SELinux TE can be used to separate security domains
- This is separation on a different layer than virtualization

"Can't we just create a user `http` and give this user file access (using UNIX permissions) to only what the webserver needs?"

Type Enforcement vs. DAC

- SELinux TE can be used to separate security domains
- This is separation on a different layer than virtualization

"Can't we just create a user `http` and give this user file access (using UNIX permissions) to only what the webserver needs?"

- There is no way in DAC to prevent another user `bdu` to make all his files readable for the webserver!
- There is no way to prevent `root` from *any* file access using DAC

MLS: Bell-LaPadula

- Central idea: control information flow
- Security model introduced in 1973
- Implemented in the Multics OS

MLS: Bell-LaPadula

- Central idea: control information flow
- Security model introduced in 1973
- Implemented in the Multics OS
- Assign to all objects *security levels*, typically:
  - **Top secret**
  - **Secret**
  - **Confidential**
  - **Unclassified**

MLS: Bell-LaPadula

- Central idea: control information flow
- Security model introduced in 1973
- Implemented in the Multics OS
- Assign to all objects *security levels*, typically:
  - **Top secret**
  - **Secret**
  - **Confidential**
  - **Unclassified**
- Assign to users *clearance levels*

MLS: Bell-LaPadula

- Central idea: control information flow
- Security model introduced in 1973
- Implemented in the Multics OS
- Assign to all objects *security levels*, typically:
  - **Top secret**
  - **Secret**
  - **Confidential**
  - **Unclassified**
- Assign to users *clearance levels*
- Assign to processes *security levels*

Bell La-Padula rules

Simple Security

A subject (user, process) must not be able to read an object above his clearance level. (e.g, a user with clearance "confidential" must not be able to read a file with security level "secret").
No read-up

The $\star$ Property

A subject (process) must not write to an object below its security level. (e.g., a process with level "secret" must not write to a file with level "unclassified").
No write-down

Tranquility

How is the security level of a process defined?

Strong tranquility

Security level of a process never changes. Set it once at startup, typically to the user's clearance level.

Weak tranquility

Security level of a process never changes the security level *in a way that it violates the security policy*. Typically start with low level, and increase as the process reads higher-level information.

Typically desirable: weak tranquility

Bell La-Padula example

- User `peter` with clearance “secret” starts process `myprog` with level “unclassified”
- `myprog` tries to read file `myfile` with level “confidential”

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential

Bell La-Padula example

- User `peter` with clearance “secret” starts process `myprog` with level “unclassified”
- `myprog` tries to read file `myfile` with level “confidential”
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level “top secret”
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level “confidential”
  - Allowed, because confidential $\geq$ confidential

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"
  - Forbidden, because unclassified $<$ confidential

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"
  - Forbidden, because unclassified $<$ confidential
- `myprog` tries to read file `topsecretfile` with level "top secret"

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"
  - Forbidden, because unclassified $<$ confidential
- `myprog` tries to read file `topsecretfile` with level "top secret"
  - Forbidden, because top secret $>$ secret

Bell La-Padula example

- User `peter` with clearance “secret” starts process `myprog` with level “unclassified”
- `myprog` tries to read file `myfile` with level “confidential”
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level “top secret”
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level “confidential”
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level “unclassified”
  - Forbidden, because unclassified $<$ confidential
- `myprog` tries to read file `topsecretfile` with level “top secret”
  - Forbidden, because top secret $>$ secret
- `myprog` tries to read file `secretfile` with level “secret”

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"
  - Forbidden, because unclassified $<$ confidential
- `myprog` tries to read file `topsecretfile` with level "top secret"
  - Forbidden, because top secret $>$ secret
- `myprog` tries to read file `secretfile` with level "secret"
  - Allowed, because secret $\leq$ secret
  - Level of `myprog` increases to secret

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"
  - Forbidden, because unclassified $<$ confidential
- `myprog` tries to read file `topsecretfile` with level "top secret"
  - Forbidden, because top secret $>$ secret
- `myprog` tries to read file `secretfile` with level "secret"
  - Allowed, because secret $\leq$ secret
  - Level of `myprog` increases to secret
- `myprog` tries to write to file `conffile` with level "confidential"

Bell La-Padula example

- User `peter` with clearance "secret" starts process `myprog` with level "unclassified"
- `myprog` tries to read file `myfile` with level "confidential"
  - Allowed, because confidential $\leq$ secret
  - Level of `myprog` increases to confidential
- `myprog` tries to write to file `topsecretfile` with level "top secret"
  - Allowed, because top secret $\geq$ confidential
- `myprog` tries to write to file `conffile` with level "confidential"
  - Allowed, because confidential $\geq$ confidential
- `myprog` tries to write to file `otherfile` with level "unclassified"
  - Forbidden, because unclassified $<$ confidential
- `myprog` tries to read file `topsecretfile` with level "top secret"
  - Forbidden, because top secret $>$ secret
- `myprog` tries to read file `secretfile` with level "secret"
  - Allowed, because secret $\leq$ secret
  - Level of `myprog` increases to secret
- `myprog` tries to write to file `conffile` with level "confidential"
  - Forbidden, because secret $>$ confidential

Extensions to Bell La-Padula

- Sometimes Bell-LaPadula is combined with categories to capture "need to know"
- Example: "nuclear", "intelligence", "submarine", "airforce"
- Compartments are subsets of the set of clearances
- Subjects and objects are assigned compartments, e.g.,
  - User `peter`: {"intelligence", "airforce"}
  - File `file1`: {"intelligence"}
  - File `file2`: {"airforce, submarine"}
- Subject with clearance compartment S is allowed to read an object with compartment O, if $O \subseteq S$
- Example:
  - `peter` is allowed to read `file1`
  - `peter` is not allowed to read `file2`

Bell La-Padula comments

- Actual write level is not defined by BL (only minimal level)
- No automated way to declassify information (i.e., reduce the level)
- In principle, users can write above their clearance

SELinux vs. Qubes

- Type enforcement can provide some sort of compartmentalization
- Very different level than virtualization from Qubes
  - SELinux cannot prevent the X-window "attack"
  - SELinux relies on kernel security
  - Multiuser approach (SELinux) vs. single user (Qubes)
- MLS mainly relevant for military applications
- Kernel is use trusted TCB, Xen is much smaller
- Different assumptions about what a compromise can do