

# OS Security

## Mandatory Access Control

Radboud University, Nijmegen, The Netherlands



Winter 2016/2017

## A short recap

- ▶ Traditional UNIX security uses discretionary access control (DAC)
- ▶ Each user decides about access permissions of his/her files
- ▶ Root can access all 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
- ▶ System-wide and more fine-grained control needed to limit scope of such an attack

# Mandatory access control

- ▶ A system implements *mandatory access control* (MAC) if the protection state can only be modified by trusted administrators via trusted software.
- ▶ Trusted administrator defines policies, for example, to determine which processes are allowed to access which files.
- ▶ Users cannot disable this.

## Multi-level security: Bell-LaPadula

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

## Multi-level security: Bell-LaPadula

- ▶ Central idea: control information flow to protect confidentiality
- ▶ Security model introduced in 1973
- ▶ Implemented in the Multics OS
- ▶ All objects are assigned *security levels*, typically:
  - ▶ **Top secret**
  - ▶ **Secret**
  - ▶ **Confidential**
  - ▶ **Unclassified**

# Multi-level security: Bell-LaPadula

- ▶ Central idea: control information flow to protect confidentiality
- ▶ Security model introduced in 1973
- ▶ Implemented in the Multics OS
- ▶ All objects are assigned *security levels*, typically:
  - ▶ **Top secret**
  - ▶ **Secret**
  - ▶ **Confidential**
  - ▶ **Unclassified**
- ▶ Users are assigned *clearance levels*

## Multi-level security: Bell-LaPadula

- ▶ Central idea: control information flow to protect confidentiality
- ▶ Security model introduced in 1973
- ▶ Implemented in the Multics OS
- ▶ All objects are assigned *security levels*, typically:
  - ▶ **Top secret**
  - ▶ **Secret**
  - ▶ **Confidential**
  - ▶ **Unclassified**
- ▶ Users are assigned *clearance levels*
- ▶ Processes are assigned *security levels*

# Bell-LaPadula rules

## Simple Security Property

A subject (user, process) must not be able to read an object above its clearance level (e.g., a user with clearance “confidential” must not be able to read a file with security level “secret”).

**No read-up**

# Bell-LaPadula rules

## Simple Security Property

A subject (user, process) must not be able to read an object above its 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 ★ 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 *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-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

- ▶ User with clearance “secret” starts process `myprog` with level “unclassified”
- ▶ `myprog` tries to read file `myfile` with level “confidential”
  - ▶ Allowed, because  $\text{confidential} \leq \text{secret}$
  - ▶ Level of `myprog` increases to confidential

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Bell-LaPadula example - weak tranquility

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

## Extensions to Bell-LaPadula

- ▶ Sometimes Bell-LaPadula is combined with categories to capture “need to know”
- ▶ Example: “nuclear”, “intelligence”, “submarine”, “airforce”
- ▶ Compartments are subsets of the set of categories
- ▶ Subjects and objects are assigned compartments, e.g.,
  - ▶ User `user1`: {“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:
  - ▶ `user1` is allowed to read `file1`
  - ▶ `user1` is not allowed to read `file2`

## Bell-LaPadula comments

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

## Biba model

- ▶ Introduced by Kenneth J. Biba in 1975
- ▶ Model to protect integrity
  - ▶ Complement of secrecy in Bell-LaPadula
- ▶ Assign to all objects and users *integrity levels*, typically:
  - ▶ **Crucial**
  - ▶ **Very important**
  - ▶ **Important**
- ▶ Prevents “pollution” of information with higher integrity level

# Biba rules

## Simple Integrity

A subject (user, process) must not read an object below its integrity level (e.g., a user with level “crucial” must not read a file with level “very important”).

## No read-down

# Biba rules

## Simple Integrity

A subject (user, process) must not read an object below its integrity level (e.g., a user with level “crucial” must not read a file with level “very important”).

### No read-down

## The ★ Integrity Property

A subject (user, process) must not be able to write to an object above its integrity level (e.g, a process with clearance “important” must not be able to write to a file with integrity level “very important”).

### No write-up

# Linux Security Modules

- ▶ 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”

# Linux Security Modules

- ▶ 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

# Linux Security Modules

- ▶ 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
- ▶ In recent kernels, hooks defined in `include/linux/lsm_hooks.h`

## Criticism of LSM

LSM is in the mainline kernel and various LSM implementations 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 implementations 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 the kernel's GPL license

## Criticism of LSM

LSM is in the mainline kernel and various LSM implementations 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 the kernel's 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 implementations 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 the kernel's 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 implementations 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 the kernel's 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

# Implementations of LSM

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

## SELinux overview

- ▶ Originally developed by the NSA
- ▶ Released as open source
- ▶ Used today by, for example, Red Hat Linux, Fedora, CentOS

# SELinux overview

- ▶ Originally developed by the NSA
- ▶ Released as open source
- ▶ Used today by, for example, Red Hat Linux, Fedora, CentOS
- ▶ Check if SELinux is enabled:

```
getenforce
```

- ▶ 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
- ▶ Released as open source
- ▶ Used today by, for example, Red Hat Linux, Fedora, CentOS
- ▶ Check if SELinux is enabled:

```
getenforce
```

- ▶ 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

- ▶ Everything (processes, files, sockets, etc) has a security context (a label) in the format:

user:role:type(:level)

- ▶ Security context for files is stored in the file system, the rest in the kernel
- ▶ Mainly important for the moment: the type

# Type Enforcement

- ▶ Everything (processes, files, sockets, etc) has a security context (a label) in the format:

user:role:type(:level)

- ▶ Security context for files is stored in the file system, the rest in the kernel
- ▶ 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 security context of network sockets

# Type Enforcement

- ▶ Everything (processes, files, sockets, etc) has a security context (a label) in the format:

user:role:type(:level)

- ▶ Security context for files is stored in the file system, the rest in the kernel
- ▶ 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 security context of network sockets

- ▶ All access has to be explicitly granted, using allow rules:

```
allow source_type target_type : object_class permissions;
```

# Type Enforcement

- ▶ Everything (processes, files, sockets, etc) has a security context (a label) in the format:

user:role:type(:level)

- ▶ Security context for files is stored in the file system, the rest in the kernel
- ▶ 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 security context of network sockets

- ▶ All access has to be explicitly granted, using allow rules:

```
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 object type (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

## 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
- ▶ SELinux can limit the damage malware or an attacker can do