Co-funded by the
Erasmus+ Programme
of the European Union

ISSES

ISSES – Information Security Services
Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant
N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

OPERATING SYSTEM SECURITY

COMPUTER SECURITY

Lecture 5

ISSES 2017-2020
Novi Sad, Serbia

Information Security Services Education in Serbia (ISSES)

5.1 Operating system security

OS as a provider of security

- When talking about operating system security it is important to realize that the OS is both the *object* of security and its provider.
- We have covered, in detail, how the CPU architecture provides certain mechanisms for the provision of security, but ultimately, these have to be configured and *used* for them to be effective.
- As a result, the OS has to do work to fully engage all the security-providing mechanisms in a CPU architecture while also providing user code with limited, supervised, controlled access to the resources it is protecting.
- In other words, a lot of the time the OS stands between the user and hardware and metes out access. The question is *how?*

OS as a provider of security

- Therefore, a crucial job of the OS is to provide some way for ring3 code to call ring0 code.
- This mechanism is known as a 'syscall' or 'system call.'
- In essence, it's asking the OS to do something on our behalf.
- From the point of view of their user, a system call is just a function (indeed, in Windows, calling suitable functions is absolutely the only way to access a system call.

System Calls—Linux

```c
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(){
    const char* s = "Hello, World\n";
    int n = 13;
    ssize_t pn = 0;
    int f = open("tmp.txt", O_WRONLY | O_CREAT);
    pn = write(f, s, n);
    if (n != pn) printf("Failed write!\n");
    else printf("Succeeded write!\n");
    close(f);
    return 0;
}
```

System Calls—Windows

```c
#include "stdafx.h"
#include <Windows.h>

int main()
{
    const char* s = "Hello, World\n";
    int n = 13;
    DWORD pn = 0;
    HANDLE f = CreateFileA("tmp.txt", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    WriteFile(f, s, n, &pn, NULL);
    if (n != pn) printf("Failed write!\n");
    else printf("Succeeded write!\n");
    CloseHandle(f);
    return 0;
}
```

Function calls?

- These might *look* like function calls but they aren't.
- After all, the code might belong to the OS, indeed, it *does* as the next slide will show, but, just JMPing into it or CALLing it will not do.
- Firstly, it's bound to use functions user code is not permitted to use, and there nothing about a JMP or CALL that changes the lastmost two bits of the CS register.
- Secondly, if the destination code is marked as ring 0 and thus allowed to use the code then, as we've mentioned in the 'memory protection' section when we discussed hardware security, then we can't even address it properly.

Writing code

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_WRITE))
                return -EBADF;
        if (!(file->f_mode & FMODE_CAN_WRITE))
                return -EINVAL;
        if (unlikely(!access_ok(buf, count)))
                return -EFAULT;

        ret = rw_verify_area(WRITE, file, pos, count);
        if (!ret) {
                if (count > MAX_RW_COUNT)
                        count =  MAX_RW_COUNT;
                file_start_write(file);
                ret = __vfs_write(file, buf, count, pos);
                if (ret > 0) {
                        fsnotify_modify(file);
                        add_wchar(current, ret);
                }
                inc_syscw(current);
                file_end_write(file);
        }

        return ret;
}
```

Wait a minute…

- That's not called 'write'
  - no, that's defined in the unistd.h and is only user-facing.
- What's a VFS?
  - The general abstraction layer for file systems which then redirects crucial FS calls to specific implementations.
- So how does me calling write lead to the execution of this function?

The Syscall Mechanism

- The CPU security architecture has methods of allowing for ring3 code to call ring0 code.
- These are:
  - Interrupts
  - Call gates
  - Fast syscalls

Interrupts

- The oldest way to do system calls is to use interrupts.
- Interrupts are already present in the kernel as a way to handle outside events.
- All that is required is to set up one, special, interrupt (typically, 0x80, i.e. interrupt 128 is used) and to set up a calling convention in which the parameters to a system call are passed through processor registers with the same method used to exfiltrate the return value.
- Then all that's required is for the user to carefully set up all the parameters and to request an interrupt be processed. This is done using the int instruction.
- But how does the system interrupt handler figure out which particular system call is called? System call number.

Example Linux syscall numbers

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c
13	rt_sigaction	sys_rt_sigaction	kernel/signal.c
14	rt_sigprocmask	sys_rt_sigprocmask	kernel/signal.c
15	rt_sigreturn	stub_rt_sigreturn	arch/x86/kernel/signal.c
16	ioctl	sys_ioctl	fs/ioctl.c
17	pread64	sys_pread64	fs/read_write.c

Syscall numbers, Windows

- Windows is a bit different: syscall numbers are *entirely different* from built-to-build.
- A user is never meant to access them.
- But that doesn't mean lists don't exist, it's just that they have to be considerably more complex. Consider CreateFile (we used it just recently) and just Windows 10.

1507	1511	1607	1703	1709	1803	1809	1903	1909
0x016e	0x0170	0x0172	0x0175	0x0178	0x0178	0x0178	0x0178	0x0178

Call gates

- Call gates are a more general mechanism.
- They are used to specify that there's a code segment somewhere in which there is a *very specific* offset to which you can JMP/CALL despite that segment having a different (higher) privilege level.
- You do so by specifying a 64-bit/128-bit (depending on whether you are running in 32-bit or 64-bit mode) entry in the GDT with the segment selector for the code to be called plus offset in the segment plus the DPL for the gate (so you can have gates accessible to some but not to others allowing for finer granularity)

Call gates

- Then, once you've set up your gate, you can JMP/CALL in far mode with the segment selector set to index the gate in the GDT, and the offset being required but not checked, i.e. it can safely be set to 0.
- Call gates can provide very fine permission granularity coupled with partial stack transfer and a number of other features.
- It's perfectly possible to implement the syscall mechanism using this, though in practice it isn't used for that. In fact, it isn't used for anything.
- Why?

Aside on CPU technologies

- When CPU manufacturers (mostly Intel) designed early PC CPUs a lot of the design was speculative, meaning they tried to build the features they thought were needed.
- Very often, this didn't turn out to be what was needed at all.
- What OSs, in particular, wanted is to do things as much as possible on their own and for everything that the CPU does to be executed simply and *quickly.*
- This is why there's a seemingly inexplicable move from segmentation with its almost baroque permission levels, gates, trap gates, interrupt gates and so on to the modern approach which only differentiates between privileged and nonprivileged code, for instance, when it comes to paging.
- And why the modern way to use system calls is much simpler.

Modern system call syntax

- Two pairs of instructions exist for this: SYSCALL/SYSENTER and SYSRET/SYSEXIT.
- Both are meant for the same thing and work in very much the same way, so we'll focus on SYSCALL/SYSRET purely randomly.
- The reason there's two is because Intel and AMD both built their own version and now both support the other one's version.
- These are instructions designed solely and entirely for the purposes of running system calls. As a result, they are fast, and only transfer control with ring0 privileges to a pre-specified syscall handler address loaded into a protected model-specific register using the WRMSR function.

Modern System-Call syntax.

```
global _start

section .text

_start:
    mov rax, 1 ;Setting the syscall number, write is 1
    mov rdi, 1 ;Setting the first parameter, the FD
    mov rsi, msg ;Setting the second paramter, the buffer
    mov rdx, msglen ;Setting the third parameter, the length
    syscall ;Calling the syscall

    mov rax, 60 ; The exit() syscall
    mov rdi, 0 ; The only parameter of exit
    syscall ;Calling

section .rodata
    msg: db "Syscall", 10
    msglen: equ $ - msg
```

The security implications

- The syscall is dwelled upon here in such considerable detail because it illustrates the security-critical parts of the User/OS/computer system.
- The OS guarantees security by essentially stopping user code from doing anything on its own and, instead, doing it for the user when asked.
- By defining (and rigidly checking the parameters of) these system calls the operating system may impose some sort of *security model* on user code.
- The hardware of the computer itself is the guarantor that the limits the OS implements are respected.

Attack surfaces

1. **The hardware.** The attacker may attempt to attack the hardware itself, causing the system which protects access to ring0 to fail.
2. **The operating system.** The attacker may attempt to confuse the operating system into granting privileges it is not supposed to, failing to implement its own security model.
3. **The user code.** A remote attacker may try to trick some application it has access to on a given system into failing *its* security model, if there is one, gaining access to either resources the application itself has been protecting or to arbitrary code execution, allowing the attacker to have the same rights in the eyes of the OS as the application itself.

Information Security Services Education in Serbia (ISSES)

5.2 THE POSIX SECURITY MODEL

A brief note on POSIX

- POSIX (Portable Operating System Interface; IEEE 1003.1-1988/2001/2008/2017) is a standard which describes what is, colloquially, called 'Unix-like' behavior.
- After all, strictly speaking, there's only one UNIX, but there's a great many operating systems which are to a lesser or greater extent attempts to reimplement it.
- To standardize all of this, the POSIX standard was formed.
- Why 'POSIX' and not 'Unix-like' or something like that? The term 'Unix' was famously the subject of vast, complex, and terrifying litigation.
- We'll be mostly be sticking to POSIX-compliant behavior but in some respects we'll falter and favor Linux since this is by far the most common POSIX-esque OS you will encounter.

A brief note on scope

- This lecture will focus on how the most basic forms of security within a system are implemented, how they operate, and what their weaknesses are.
- Advanced forms of security and how the security of a POSIX system may be increased is something that will be covered in later lectures, insofar as possible.
- For now, the crucial takeaway is to achieve a mental map of the limits of the POSIX security model, as most commonly implemented and used.

Basic access control in POSIX

- The basic form of POSIX access control (i.e. who can do what, when) is based on certain core facts:
  - Anything being done in userland on a POSIX system is being done by someone (some user in some group).
  - Any file or process in the system must have an owner (user, group again)
  - Any file or process being created is owned by whoever created it.
  - The owner of the process or file has control over it.
- This is augmented by the existence of a special user ('root') who has two powers:
  - It acts as if it is the owner of everything on the system.
  - It has access to a family of restricted APIs for system management, generally this refers to restricted system calls.

File reliance

- Because of the general abstraction characteristic of POSIX systems which treats as much as possible as files, the fact that files have owners and a set of rules defining access is crucial.
- This means that, for instance, one major way the kernel both bars and allows access to hardware is by finessing the permissions in the /dev virtual file system.
- This allows, even using the relatively crude permission system, for surprising granularity in granting access control.
- More on how this can be used later.

File access control, basic

- The basic file access control is based on setting for each file the following attributes:
  - Who owns it?
  - What its access matrix is.
- The access matrix consists of zeros and ones (not permitted, permitted) with one axis being the permission (read, write, execute/scan) and the other describing the three possible relationships between the identity of the user and the identity of the owner: U (identical), G (same group) and O (other).
- When multiple matches are possible (i.e. the owner of the file is by necessity in the U, the G, and the O relationship) the most specific permission takes precedence.

File access control, basic

- This file access control is often described not as a matrix but as a nine-bit value comprising of three three-bit (RWX) groups corresponding to U, G, and O.
- This, in turn, is represented through three octal values, with, e.g. 755 being equivalent to all permissions to the user (111), and only read and execute for the G and O.
- The reason for this is historical.
- These three octal values are occasionally expanded by one more, special one whose bits, in order, least to most significant are 'sticky,' 'setgid,' and 'setuid.'

Read/write permissions

- These operate as can be expected: read allows the contents of the file to be read.
- Write, on the other hand, allows the contents of a file to be modified, appended to, or truncated.
- Deleting the file, however, is not possible with write permissions. What is required is write permissions to the *parent directory.*
- This may seem counter-intuitive but fits perfectly with how, say, the ext family of filesystems implements files and directories.

Execute/scan permissions

- When applied to a file this is the permission to execute which can be done on binary files (typically in the ELF file format) or on text files which may be executed directly provided they specify a command interpreter in a special line at the start which starts with a #!.
- When applied to a directory, it allows a directory to be traversed as part of some larger path but does not imply the right to list its contents or modify them.
- For this, the appropriate read/write bits must be set.

Setuid/Setgid

- Executable files with the setuid or setgid bits set when executed don't belong to the user/group of the identity responsible for them being started but instead get the user/group the *file* belongs to.
- When used on a directory the setgid bit has a completely different meaning: any files made in the directory won't belong to the user's default group (as normal) but instead to the group that owns the directory.

The sticky bit

- Computer terminology can be, on occasion, *very peculiar.*
- It's use on files has no effect: historically it was used for executable files in early UNIX systems.
- It is still used on directories, however: a 'sticky' directory won't allow the renaming or deletion of a file within it unless it is done by:
  - the owner of the sticky directory,
  - the owner of the file, or
  - the superuser/root.
- The chief use for this is to make directories like /tmp more secure: this way everyone can make their own files in /tmp but can't interfere with the files others make.

Ownership of processes

- Whoever starts a process, normally (but see setuid/setgid earlier) owns the process and may:
  - send the process controlling signals, or
  - alter the scheduling priority of the process, provided the change reduces the priority.
- The superuser overrides these rules and, additionally, may *increase* the priority of any process.

The root/superuser account

- The unique user ID of 0
- Is not subject to most controls of permission, normally.
- Some things are still forbidden as mistakes, such as running a process with no execute permission.
- Of course nothing stops the root user from *altering* the permission. This is not a security as much as correct operation matter.
- The superuser may perform operations unique to it: changing the system date/time, creating device files, modifying priorities and resource controls, and other system administration tasks.
- Processes that the superuser starts may also (uniquely) change their uid and gid to any value, losing their privileged status.

Point of weakness: Root

- The root/superuser account is a significant single-point of failure.
- Such a user *must* exist, but each use of those absolute permissions is in a sense, risk.
- Gaining root access is the ultimate goal of subverting operating system security.
- For this reason, the root account is often the subject of fairly stringent security policy.

Typical root limitations

1. A very, very strong password.
2. Disabling login from anywhere using any means short of a login at the system's physical hardware console.
3. Disabling login *altogether* and requiring that the user first login as a normal user and then elevate their privileges to a higher level by executing the 'su' command and providing the root password.
4. Disabling the root account entirely and, instead, parceling out access to its powers using the 'sudo' command.

Point of weakness: su

- su (allowing anyone with the appropriate passwords to switch their identity to a different user) and other software that permits logging in (see also sudo, later) are just normal commands.
- This causes a security vulnerability: the user has no ready method to identify that they have run the version of the program that they want instead of one written by a malicious attacker and injected into their path (modifying a user's command search path is not a sensitive task).
- This means that it's not difficult to, with momentary access to someone's account create a password-harvesting trojan-horse which is difficult to detect.
- This is why Windows still hides security-crucial actions behind CTRL-ALT-DEL. User software can intercept that.

sudo

- The sudo command is used to execute a single command with elevated privileges: formally as any user, but typically as root.
- sudo operates by:
  - Verifying that the user is allowed to perform sudo actions at all (using /etc/sudoers)
  - Verifying that the command is listed as permitted for that user (also in /etc/sudoers, allowing limited abilities to be permitted to selected users)
  - Verifying the identity of the user one more time by asking them to retype their password.

sudo

- Sudo allows for considerable granularity of permissions since abilities can be divided by command.
- However it has weaknesses: if sudo can be allowed to get to a shell it's protection collapses.
- While it is possible to *forbid* certain commands all this does is forbid commands *by their path.* Copying /bin/bash to a suitable location and running it from there is not something that can be conveniently stopped.
- This means that, while powerful, sudo has limits.
- Furthermore, while it does lessen risk vis-à-vis what a single user can do, the use of sudo does expand the attack surface of a system to cover any user listed in sudoers, and as is well known, security is only as good as its weakest link.

Faults of the basic security model

1. Limited provision for network security
2. No way to modularize permissions without expanding the attack surface of the system, either through sudo-commands or setuid commands.
3. The root account as a single point of failure.
4. Limited expressivity of group-based sharing.
5. Limited expressivity of hardcoded policy-decisions.
6. Limited logging facilities.

Fixing the standard model

- The way the standard model is fixed is to either:
  - Add additional software which operates with the standard model to expand it in some way, usually accomplishing some specific task.
  - Modify the OS kernel itself with new functionality.
- We'll briefly cover some advanced methods of securing a POSIX-type OS
- Some of these will be covered in greater detail later, but the subject is easily broad enough to exceed the boundaries of this course.

Access Control Lists

- ACLs are a way of improving the permission system protecting files in a way which allows for more flexibility.
- The key idea behind ACLs is that they are flexible: each ACL describes the rules of access for a single file and is comprised of a flexible number of ACEs—access control entries.
- An ACE specifies either a user or a group and then permissions which apply for that user or group.
- This is (as we'll see later) exactly how Windows does things.
- This seems like the obviously correct way to do things. Is it?

Not really

- More isn't always better.
- ACLs are definitely more flexible and more powerful, true, but practice means that complex ACLs are also more prone to bugs.
- In essence to enable complex sharing rules (which are needed for what is, in truth, a very small subset of files) a vast new attack surface of misconfigured and ambiguous ACLs is introduced.
- Linux administration best practices generally recommend two use-cases for ACLs:
  - An absolute need for filesystem-level complex access control (very rare)
  - Compatibility with Windows file systems (very common)

Types of ACLs

- ACLs exist in two mutually incompatible forms
  - POSIX ACLs
  - NFSv4 ACLs
- POSIX ACLs are a bit of a misnomer because, while *inspired* by work POSIX did to specify ACLs, there was never a standard. Nevertheless POSIX ACLs, are fairly standardized and operate as a *de facto* standard. They are also the only ACLs supported on Linux.
- NFSv4 ACLs grew from an attempt for the NFS file-share protocol to support fully interoperability with Windows and its somewhat more elaborate permission system.
- As a result NFSv4 ACLs are an attempt at forming a superset of all ACL systems and are, as a result, fairly complex.

ACL implementation specificity

- Because there's multiple standards and because permissions require the cooperation of system software (which exposes methods for manipulating ACLs), the filesystem (which must store the ACLs), and the kernel (which must support all this interoperation) the situation with ACL implementation can get… troublesome.
- Consider the ZFS file system: if created and used on its native Solaris it uses NFSv4 as default, however its Linux port uses only POSIX ACLs. Thus it is possible to mount a partition built with one system on another system and try to manipulate it with invocations of the chmod command which know essentially nothing past nine-bit UGO access control.

Point of weakness: frustrated admins

- Situations such as these can cause repeated problems.
- The most visible form of problem caused by security and access control subsystem malfunction is lack of permission to do something the user should be able to do.
- Enough of such problems may provoke the people administering the system to allow for overbroad permissions just to get whatever the system is meant for to work.
- These may be justified as 'temporary' or 'just for testing' but bitter experience shows that nothing's more permanent than a 'temporary' security hole opened up so vital production code will run.

POSIX ACLs

- POSIX ACLs (hereafter PACLs) are fairly straightforward extensions of the system we already described.
- The permissions being manipulated are the same as in the 9-bit world of the UGO/RWX matrix: read, write, execute.
- The difference is that PACLs allow these RWX bits to be set for any combination of users and groups.
- The syntax for ACEs within a PACL is one of 'user', 'group', 'other', or 'mask' followed by a colon followed by an identifier of the group/user or nothing. Nothing is mandatory in the case of 'mask' or 'other' and in the case of user/group it signifies the user/group of the owner. Then there is a colon followed by the access specifier which consists either of the letters 'r,' 'w,' or 'x' in that order or the character '-' indicating a lack of permission.

POSIX ACE examples

- **user::rwx**
  - The file owner may read, write, or execute the file.
- **user:veljko:r--**
  - The user 'veljko' may only read the file.
- **other::---**
  - Users not otherwise specified can't do anything with the file.
- **group:admin:rw-**
  - Users in group 'admin' may read and write the file.

Backwards compatibility

- PACLs and traditional 9-bit permissions must never conflict
- Therefore alterations of one, modify the other so that they are not in conflict.
- This is fairly complex and is based on some basic rules:
  - user:: and other:: are exactly the same as the U and O bits.
  - The owner always has whatever is given by user::
  - Users not specified in any other way have what's specified in other::
  - Files without ACLs or files with ACLs that have three ACEs, user::, group::, and other:: are exactly equivalent to the 9-bit UGO permissions.

The mask permission in PACLs

- The last rule is the most complex and concerns the mask:: permission.
- The mask permission is mapped bidirectionally to the 'group' level of permission in the 9-bit model.
- Further, the mask:: permission has a special additional feature: it functions as the upper limit on the permissions that an ACL may grant on any:
  - user by name
  - group by name
  - default group (i.e. group::)
- Why? Well the U and the O are not a concern and with this setup the G can claim more permissions than are actually present (since it is an upper bound) but never *less* than what's actually present.

Defaults

- PACLs for directories may have a series of ACEs marked 'default.' These will be applied to any new files made in the directory.
- These will also be copied *as defaults* into any subdirectories being made.
- This means that defaults can propagate down subdirectories.
- This link is not maintained. The only important thing is what the defaults were at the time of subdirectory creation.

NFSv4 ACLs

- NFSv4 ACLs (hereafter NACLs) are, very nearly a direct port of Windows ACLs (more on them later)
- The differences between PACLs and NACLs are:
  - The syntax is slightly different (owner@ instead of user:: for instance)
  - They have a *lot* more permission bits (see next slide)
  - Instead of using defaults, any ACE within a NACL may be marked as being inheritable
  - There's no mask.
  - ACEs are partial: aside from specifying who and what they specify either 'allow' or 'deny' and multiple ACEs may be applied to one request with the precedence determined by order. This means that it is possible to get back an 'I don't know' to a permission request. This blocks access.

NFSv4 file permissions

Code	Name	Meaning
r	read_data	Read file data/list entries in a directory
w	write_data	Write data to a file/add file (not directory!) to a directory.
x	execute	Execute, much like the 'x' bit in the UGO system
p	append_data	Append data to file or add subdir (not file!) to a directory
D	delete_child	Delete item within a directory. Does nothing to file.
d	delete	Delete file/directory.
a	read_attributes	Read normal attributes.
A	write_attributes	Write normal attributes.
R	read_xattr	Read extended attributes stored as key-value pairs.
W	write_xattr	Write extended attributes stored as key-value pairs.
c	read_acl	Read ACL
C	write_acl	Write ACL
o	write_owner	Change ownership (normally reserved for root)
s	synchronize	Allow requests for synchronous I/O. Obscure.

Linux capabilities

- Instead of having an omnipotent superuser account, systems implementing the Linux capabilities subsystem provide a set of about 38-ish 'capabilities' each of which is a specific permission to do a specific type of task usually reserved for root.
- This has potential despite being an implementation of a POSIX standard that was never adopted as a standard and not being a capability system at all.
- A root user/process merely has *all* capabilities so there's direct compatibility and a fair bit of granularity.
- Rarely used in practice Linux capabilities are instead used to implement more complex security-providing applications.

Linux namespaces

- Linux namespaces are a tremendously powerful feature which, instead of creating an increasingly baroque filigree of rules simply carves out a subset of the system's resources (including files and processes, but also ports) and confines a process to just those.
- Within their restricted domain the process may even run as root: it doesn't matter. What it can see is what it's been given. Everything else is completely isolated.
- This has been accomplished by giving each process multiple PIDs associated with each namespace in which it runs. (Namespaces can nest)
- This means that a process may see itself as a PID=1 Init-type process in full control, while the process which spawned it sees it as a normal process with a normal PID.

Linux namespaces

- This is implemented through calls of the 'clone' and 'unshare' system calls.
- In practice Linux namespaces are not used directly but are often combined with capabilities (and a host of other techniques) to implement containerization functionalities like Docker
- Containerization is a magnificently effective way to secure systems provided, of course, it is used carefully.
- Containerization and sandboxing in security is a bit out of scope for this introductory course, but will be a subject of some additional remarks and examples in the lecture on OS hardening.

Kernel plugins for additional security

- There's only so far the POSIX model can be pushed.
- For when more is required, both Linux and BSD permit the kernel to be enhanced with pluggable kernel modules which expand the system with entirely different security modules with support for mandatory access control, and role-based access control.
- These modules (plus their associated userland software) such as SELinux and AppArmor will be the subjects of, to varying extents, of lectures 6 and 7.

Information Security Services Education in Serbia (ISSES)

5.3 POSIX USER MANAGEMENT AND AUTHENTICATION

User management

- Everything in userland on a POSIX system is, in some way, associated with some user.
- Users on a given POSIX system are identified with
  - UID (for the system)
  - username (for the user)
- The UID is for internal use only while the username is how a user logs in.
- Aside from the UID, the user has attributes both functional and related to security.

User API

- The way a coder would get information about a user is covered through the API of POSIX operating systems.
- For instance getpwuid() takes and UID and returns the user's full record.
- getpwnam() is how to do the same thing with a username.
- This is not only useful but allows for *pluggable* methods for keeping track of users.
- The API determines a single point of reference for the programmer: the system library and its functions.
- The OS can then redirect those requests to all sorts of subsystems using the /etc/nsswtich.conf configuration file.

User attributes

- The simplest and default way to keep track of users is the /etc/passwd system
- Details about a user in a POSIX system are typically in /etc/passwd, with one user per line, and the fields separated by colons. These fields are:
  - username
  - password [now never kept in the /etc/passwd file, instead replaced by a single lowercase 'x']
  - UID
  - The default group to which a user's files will be credited
  - GECOS field: misc. data about the user
  - User home directory
  - Optionally the user shell/command interpreter

Point of weakness: UIDs

- UID collisions may be a security vulnerability in the case of an integrated system of multiple computers managing a somewhat shared set of files.
- In this cases it should be noted that the system has absolutely no way to identify ownership over a file aside from noting the UID of its owner.
- Collision in the UID space means potential misidentification of an owner.
- To help with this, the Network File System maps remote root users to a special, system, 'nobody' user in order to stop remote privileged users from controlling systems that aren't their own.
- This is nearly universally solved through centralizing user databases. More on this later.

User attributes

```
test:x:1001:1001:Test Testenson,,,:/home/test:/bin/bash
```

Note 1. This screenshot, much like all POSIX screenshots in this lecture material, captured off an instance of Ubuntu Linux 19.10 running Linux Kernel 5.3.0 in x86_64 mode.

Note 2. The section with the user's name has a number of *comma* separated fields. These are somewhat archaic and never used nowadays, but are meant to contain personal details about the user. We will not concern ourselves with this section.

Regarding the password

- It used to be the case that the password field in /etc/passwd was actually used.
- This posed certain difficulties since it was visible to anyone logged into the system since /etc/passwd was also the database of everything public about the user, too.
- This was solved using password encryption: passwords were stored in obscured form and later in *hashed* form.
- In theory, this helped.
- In practice, this just helped spur the invention of password cracking software.
- Thus, all modern Linux distributions have long-since switched to using /etc/shadow (FreeBSD calls the same file /etc/master.passwd)

/etc/shadow

- The shadow file contains the (still encrypted; technically speaking hashed and salted using the SHA512 hashing algorithm in modern systems) passwords in a structure similarly separated by colons
- The fields are:
  - username
  - hashed/salted password
  - date of last password change (as days since epoch)
  - minimum password age
  - maximum password age
  - password warning period
  - password inactivity period
  - account expiration date

Password hashing types

Prefix in /etc/shadow or equiv.	Algorithm
1 or $md5$	md5
2	Blowfish
5	SHA-256
6	SHA-512

Point of weakness: Hashing

- Hashing algorithms have a distressing tendency to fail.
- In recent memory the md5 and SHA-1 hashing algorithms have both been rendered insecure through advances in cryptanalysis.
- As a result an important weakness (or a point were defense should be created) is the hashing algorithm used for the passwords: there isn't really a standardized place where it should be changed (some systems use /etc/login.defs, some use PAM (which we'll touch on) some use other means) nor a standardized procedure.

Point of weakness: Passwords

- Passwords are frequently the biggest weakness in most systems that use them.
- Two important tools exist for administrators to manage (or fail to manage) the risks of passwords:
  - A consistent, enforced password policy.
  - Procedures on what to do in irregular situations (i.e. when doubting that the user's passwords have been leaked).
- An attacker can use failure to properly implement either or both to gain an advantage.
- Most POSIX systems have some method of defining a concrete policy regarding the *form* of the password allowing things such as a minimum number of characters to be set.
- Do note that with modern systems *password length* is *vastly more important* than password complexity.

Groups

- Though they have been designed originally as an accounting measure groups in modern POSIX systems are meant to facilitate sharing files between users.
- Each groups is defined by a unique number, the GID.
- The file /etc/group defines the group ID, the name of the group, and the members of that group.
- A user is counted as a member of *all* of their groups, the default GID field in passwd is merely there to determine which group a user's files belong to, by default.
- It's today common practice to assign to each user a default GID of a group they are the sole member of which is to say that if a user wishes to share a file the user must indicate so explicitly.

Point of weakness: Lockout

- It may become necessary to temporarily prevent login by a user to a system.
- A typical use-case is in the case a user's login is suspected to be compromised and their files need to be preserved (for forensics purposes if nothing else).
- This is surprisingly difficult to do correctly.
- Modifying the password field so that it starts with a non-regulation character (! is popular) seems a direct approach and it is true that doing so either manually or through the usermod -L command (on Linux) or pw lock (on BSD) will prevent the password check from passing.
- However, there are methods of logging in that do not, necessarily, touch the password field. SSH is the most prominent.

Point of weakness: Lockout

- The alternative to this approach is to set the user's shell to something that fails instantly (/bin/false is popular) and is too simply implemented to mount an attack against.
- This is efficient because without a shell the user can't do much.
- The problem is what to do over systems which check the user list for access but don't actually launch a login shell. From GUIs to FTP there's quite a lot of things that don't really need to launch BASH to get work done.
- Sometimes these things check to see if the shell is one that's used for stopping login access (sometimes by checking /etc/shells) and sometimes they do not.
- If insufficiently tested, this may pose a security vulnerability that's easy to exploit.

Pluggable Authentication Modules

- The system for user management as described is very archaic and would be unacceptably difficult to implement security policies for.
- The problem is that, as described, there's code in 'login' and 'passwd' and tools like that which verifies passwords and sets them up and not much that can be done to change matters, not without writing our own code.
- PAM changes that by implementing a shared library that all software that deals with user authentication must use.
- This allows for a single place where
  - Improvements to authentication technology may be implemented, as needed.
  - Policies regarding authentication may be configured.

How is PAM configured?

- PAM config is a series of lines each of which engages for some purpose one of the modules that are a part of PAM.
- The syntax of a line is a whitespace-delimited list containing:
  - The type of the module (Determines what the line is used for: auth logs the user in, account limits how the login may take place, session contains actions which must take place before or after a user's session with the system, and the password modules are responsible for enforcing password-changing rules)
  - The control flag (Determines how modules within a stack interact)
  - The path to the module
  - Optional arguments to the module

The control flag

- include
  - Actually only used to include another configuration file
- optional
  - Won't affect stack execution either way: on success or failure.
- required
  - Failure causes the stack to fail in its entirety but only after everything else has had a chance to run: this prevents a side-channel attack.
- requisite
  - Fails immediately.
- sufficient
  - Isn't really sufficient, not quite. If a sufficient module succeeds, the stack stops executing, but a failure caused by required modules failing can't be overridden.

Modules

- The key feature of PAM is, of course, modules.
- For instance, if login via fingerprint is available (it is, as it happens) it can be implemented as a separate module and then placed in the appropriate user-authentication stack.
- This is how PAM may be used to extend the capabilities of Linux login without demanding that legacy code be modified.
- This is also how PAM may be used to integrate the use of *distributed* authentication and authorization solutions.
- A daemon called System Security Services Daemon (sssd) runs in the background and communicates with distributed solutions for authentication and authorization.

Distributed authentication

- In a situation with multiple networked computer systems it is very common to wish that users can log in using the same login details throughout the system.
- For one, this helps obviate a lot of the issues with collisions and for another it is both more secure and convenient, a rare and precious combination.
- The way this is done is to have a database of user account details and to have some sort of centralized system to guarantee user identity.
- The details of how this works are out of the scope of this course (which is why you have network security)

LDAP

- *The* protocol for distributed user data (a sort of /etc/passwd and /etc/shadow kept on servers and made available through a wide araa) is the Lightweight Directory Access Protocol
- LDAP is implemented by both open-source distributions such as OpenLDAP, and is the default protocol for Microsoft Active Directory in a rare case of OS interoperability.
- LDAP's chief purpose is to store and distribute data. While it is possible to authenticate against LDAP it's not uncommon to use a robust effective authentication solution called Kerberos (again something that is found both in Linux and Windows spheres)

Kerberos

- Kerberos is a network service which knows how to (securely) accept passwords and use them to grant users cryptographic tokens which can be used to authenticate the user to various services, computers, etc. on the network without demanding that those services know the user's password.
- Kerberos is a MIT-developed protocol which has also been implemented as part of Active Directory.
- Thanks to this and the pluggable authentication of PAM it is possible to have Linux and Windows networks share a surprising amount of network infrastructure.

Information Security Services Education in Serbia (ISSES)

5.4 The Windows security model—policies and differences

A note on scope

- The Windows section will focus on a few details of the way Windows security works.
- Of particular interest is how Windows and Linux differ in their respective security models
- The Windows section is somewhat narrower in scope because Windows has, essentially, one way to work with security and because a lot of the introductory work has been done already.
- For instance file security, as part of Windows has already been described when describing NFSv4 ACLs. Likewise, users and groups also exist and operate through similar means.
- What will be focused on, instead is a feature of Active Directory known as *Group Policies.*

Active Directory

- The way security is designed to work on Windows is that there's a centralized database of every object that needs to be managed in a network installation.
- This is a LDAP database and it is maintained by a service known as 'Active Directory.'
- Active Directory holds information on all computers, all user accounts on all computers, and crucially the information on *policies.*
- When not running in a network environment a local version of the Active Directory database contains all relevant information.

Group Policies

- The Windows security model is in many respects like the POSIX one
- Certainly, the way files are secure is a more granular form of ACL, but the central notion of users owning files and having rights to them remains.
- The key difference is the idea of highly granularized modifications to the rights of users and capabilities of machines known as 'group policies.'

Group policy terminology

- **Group policy**
  - The general term for the whole idea of active-directory stored rulesets applying to, potentially, large number of machines and users.
- **Policy setting**
  - An individual rule allowing, disallowing, or configuring something.
- **Group Policy Object**
  - A container for an arbitrary number policy setting stored within an Active Directory Domain Controller
- **Group Policy Preferences**
  - Extended group policy functionality
- **Preference Item**
  - A GPPref equivalent to a Policy setting.

Division of policy objects

- Each policy object divides into two branches:
  - User
  - Computer
- This means that GPOs apply to what limits/controls the user and to what limits/controls the computer in question.
- Windows further subdivides policy into a hierarchical structure of folders and subfolders. For instance a policy's User segment divides into Software Settings, Window Settings, and Administrative Templates, with the latter dividing further, into Windows Components, System, Network, and so on.
- At the bottom of this subdivision hierarchy are invididual settings.

Rule collision

- Generally the sets of Computer and User settings are disjunct.
- Where they overlap it is possible to have conflicting rules set up.
- In this case the machine rules generally override user rules where such conflict is possible: i.e. there may be a GPO which matches to a user and allows a user a certain kind of access but if said access is disabled on a GPO which matches to an individual computer and the user is operating on said computer then the action will still be blocked.

Local group policy

- While group policy is very much intended for network environments and to be what's formally known as *domain-controlled group policy,* equally present is s the single-computer version: *local group policy.*
- Any windows machine past Windows 2000 has them and editing them is trivial, just launch GPEDIT.MSC
- Local changes function much like there being a single GPO for the entire machine that's auto-applied and that GPEDIT.MSC modifies.
- With an Active Directory GPOs can be multiple (stored in an AD) and 999 at a time can apply to a user/machine.

Targeting GPOs

- All resources managed as part of an AD flow through three organizational levels:
  - Site
  - Domain
  - Organizational Unit
- OUs can be indefinitely nested one within another.
- A GPO is defined at one level or another and applies at that level and all sublevels.
- The total policy of a given object consists of the accumulation of all the GPOs that have inherited it.
- In cases of conflict, as is customary, more specific rules win with the exception of local group policy which is subordinate to AD policy.

Storing GPOs

- All GPOs are stored in what is, in essence, one large table and are 'moved' to OUs or sites or whatnot through linking
- The mechanism is very similar to the way, in a ext-family file system, that all files are together in one big inode table and are placed in folders only by the creation of hard links.
- This allows (by linking one GPO to multiple places) to effectively share GPOs between multiple target groups.

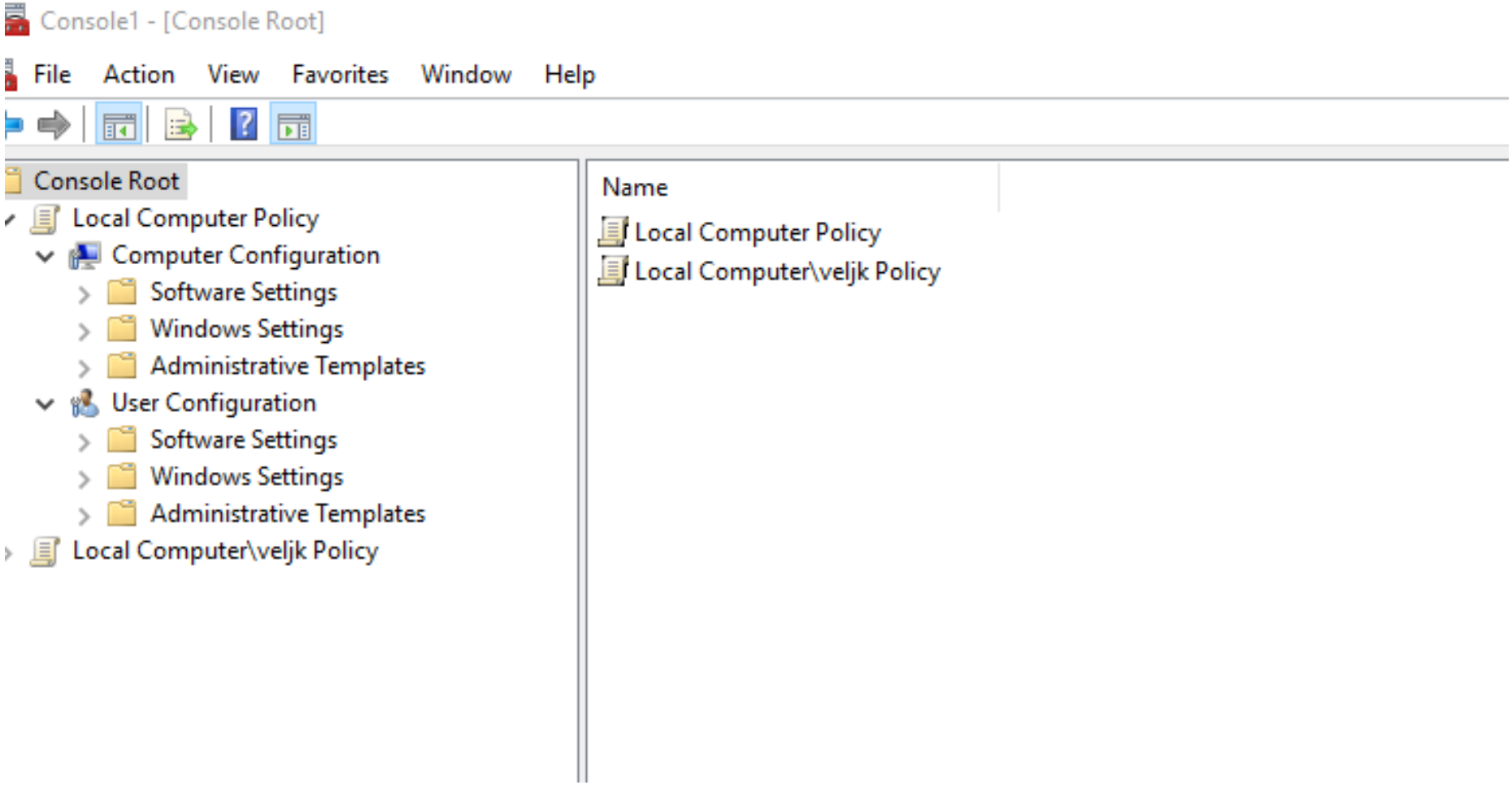
Multiple local GPOs

- It's technically possible to have multiples of local GPOs on Windows OSs past Vista.
- This is done by creating multiple layers within a machine starting with L1 the local policy that's still singular and universally applied to the machine and all its users. The L1 layer is also the only place where machine-specific settings may be placed.
- L2 has a higher priority and can discriminate between admin users and non-admin users.
- L3 targets an individual, specific user. It has the highest priority.
- The aggregate GPO applying to a specific user is the sum of all GPOs with conflicts being resolved based on priority.

Editing multiple local GPOs

- More on this in the lab, but in brief:
  - Run MMC
  - Add/Remove Snap/in
  - Add the GPO Editor Snap/in
  - Then, in the configuration, browse to select the specific GPO-category you want.
  - The end result looks like this

Editing multiple local GPOs



Policy — example

- Password policies are only possible in POSIX systems by making use of PAM and configuring modules in the password stack.
- How does a similar configuration look under Windows?
- Under 'Local Security Policies' find 'Security Setting' then 'Account Policies' and 'Password Policy' and within that there's a collection of settings regarding passwords including password aging and minimum password lengths and complexity requirements

Windows password complexity

- Not contain the user's account name or parts of the user's full name that exceed two consecutive characters
- Be at least six characters in length
- Contain characters from three of the following four categories:
  - English uppercase characters (A through Z)
  - English lowercase characters (a through z)
  - Base 10 digits (0 through 9)
  - Non-alphabetic characters (for example, !, $, #, %)
- Complexity requirements are enforced when passwords are changed or created.

(Direct quote from Windows 10)

Policy—Granularity

- As an example of granularity: within User Configuration -> Administrative Templates -> System it is possible to limit the user to only specific applications or to disable completely the access to the command prompt.
- This last one is particularly interesting as it is impossible to really do in Linux and seriously damages the ability of any attacker who has access to a low-security workstation to escalate their privileges.

Difference in approach between POSIX and Windows

- POSIX's core security model is focused on regularity, simplicity, and the building up of more complex systems from predictable, simple parts.
- This can be used to make an incredibly flexible security system while still retaining the core of simple predictability.
- But this also means that security policies that, from the point of view of the user are trivially simple (disallow burning of optical media for users X, Y, and Z) turn out to require a tremendous amount of building up.
- In the Windows model it's the job of a single setting set in an appropriate GPO.
- This seems better, and for some things: it *is.*

Point of Weakness: Complexity

- The problem with an incredibly sophisticated set of tools like the GPO subsystem is that it takes serious effort to implement good security.
- The default (the only reasonable default) of all GPOs is 'no configuration.'
- This means that every single block requires a conscious effort to produce.
- This, in turn, means that to produce an environment of minimal privilege (where the user has all the rights they need to get their work done but absolutely nothing over that) requires a lot of expensive trial and error, given that any failure will manifest itself as the security system breaking workflow.

Point of Weakness: Complexity

- Thus, a well-secured Windows environment is remarkably resistant to privilege-escalation attacks.
- A poorly-secured Windows environment has innumerable attack surfaces.
- Most Windows installations are not configured well or tightly enough.
- This will be further elaborated on in the lecture on hardening.