


Advanced Operating Systems
MS degree in Computer Engineering
University of Rome Tor Vergata 
Lecturer: Francesco Quaglia

Software security aspects

1. Classical software vulnerabilities
2. Authentication and habilitation
3. Protection domains and secure operating systems
4. Reference Monitor architectures

IT security: the very baseline

1. Systems/applications must be usable by legitimate users only
2. Access is granted on the basis of an authorization, and according to the rules established by some **administrator (beware this term)**
 - As for point 1, an unusable system is a useless one
 - However, in several scenarios the attacker might only tailor system non-usability by legitimate users (so called DOS – Denial of Service-attacks)

DOS basics

- Based on flooding of
 1. Connections (TCP layer) and (probably) threads
 2. Packets (UDP and/or application layers)
 3. Requests (on application specific protocols)
- In some sense these attacks are trivial since they could be typically handled by trading-off operation acceptance vs current resource usage
- However the big issue is how to determine what to accept (and what to reject) in the flood
- Rejecting all at a given point in time would lead to deny the execution of legitimate operations

Overall

- Copying with DOS is **not exactly** a matter of how to build system software
- It is essentially a matter of how to identify “good” things in the flood (we need methods!)
- Clearly, the identification needs to be done on the fly in an efficient manner
- So we need anyhow mechanisms for making the software performing the identification task scalable
 1. Multi-core exploitation
 2. NUMA awareness
 3. Non-blocking parallel algorithms ...

Let's slide to the “legitimate” term

- This term includes a lot of IT concepts, mostly related to the **access to resources**:

➤ Memory contents

➤ Code or portions of it (either user or kernel)



The very bad part since it can imply the memory content illegitimate usage scenarios

Security approaches

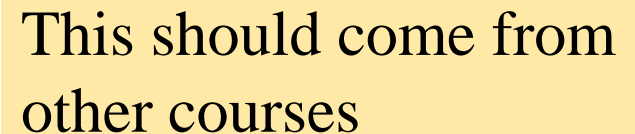
- They are typically 3
 1. Cryptography (e.g. for memory contents)
 2. Authentication/habilitation (e.g. for code or portions of it, including the kernel code)
 3. Security enhanced operating systems (as a general reference model for system software configuration and resource usage)
- Each approach targets specific security aspects
- They can/should be combined together to improve the overall security level of an IT system

Non-legitimate access to memory contents: what we looked at so far

- Side channel
- Branch miss-prediction (Spectre variants)
- Speculation along “trap” affected execution paths (Meltdown)
- Speculation on TAG-to-value (LT1 terminal)
- Hacked kernel structures (sys-call interface, VFS operations ...)

The countermeasures (so far)

- Randomization (of the address space and of data-structure padding) – compile/runtime
- Signature inspection (avoidance of dangerous instructions for data/code integrity) – loadtime
- Cyphering – runtime
 - For streams
 - For device blocks
 - For memory pages/locations



This should come from other courses

Password cyphering

- One via the `crypt()` standard function
- Works with
 - Salt
 - Different one-way encryption methods

<u>ID</u>	<u>Method</u>
-----------	---------------

<u>1</u>	<u>MD5</u>
----------	------------

<u>2a</u>	<u>Blowfish (on some Linux distributions)</u>
-----------	---

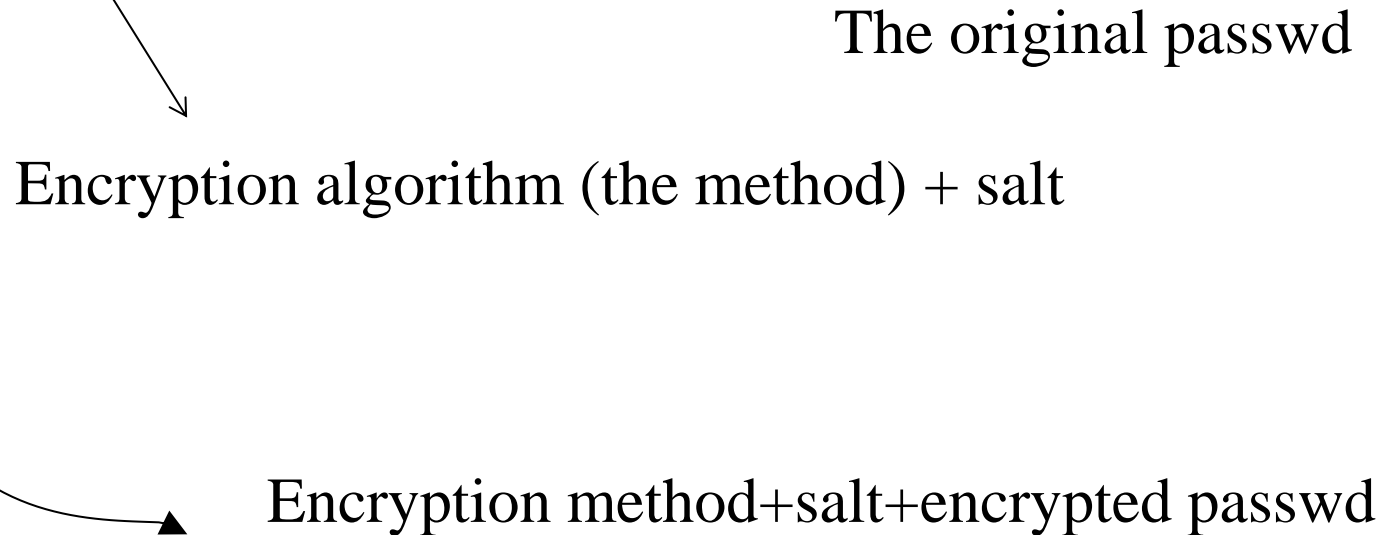
<u>5</u>	<u>SHA-256 (since glibc 2.7)</u>
----------	----------------------------------

<u>6</u>	<u>SHA-512 (since glibc 2.7)</u>
----------	----------------------------------

Encryption library function

```
#include <unistd.h>
```

```
char *crypt(const char *key, const char  
            *settings)
```



Lets' look at UNIX (Linux) systems

- The passwords' database is kept within 2 distinct files
 1. `/etc/passwd`
 2. `/etc/shadow`
- `/etc/passwd` is accessible to every user and is used for running base commands (such as `id`) - BEWARE THIS!!
- `/etc/shadow` is confidential to the root user, and keeps critical authentication data such as the encrypted passwords

Non-legitimate access to code: what we looked at so far

- Miss-speculation (for branches or traps)
- Hacked kernel structures (sys-call interface, VFS operations ...)
- Hacked hardware operation mode

The countermeasures (so far)

- The same as before, plus ...
- Explicit value corrections on branches (see the syscall dispatcher), ...plus
- full avoidance of kernel modules insertions (which could otherwise subvert all the used countermeasures)!!

The big questions here is: who does the job of mounting a kernel module??
A human or a piece of code??

The answer is easy

- If no thread is active, then no module load can ever take place
- If there is at least one thread active in the system, then the answer is clearly: **a piece of code that can be run along that thread**
- So, what if we make non-legitimate usage of a piece of code along an active thread??

Coming to buffer overflow

- It is a mean for leading a thread to make non-legitimate usage of memory locations, including, blocks of code
- These blocks of code can already be present into the address space accessible by the thread
- Or we can inject them from an external source
- Or we can compose them by fractions we take somewhere

The technical point

- A buffer overflow leads the content of some memory location to be **overwritten by another value**
- The newly installed value is however non-compatible with the actions that a thread should perform based on its control flow logic
- Minimal damage: e.g. some segfault
- Maximal damage: the thread grubs access to any resource (code/data in the system)

Lets' begin from the beginning

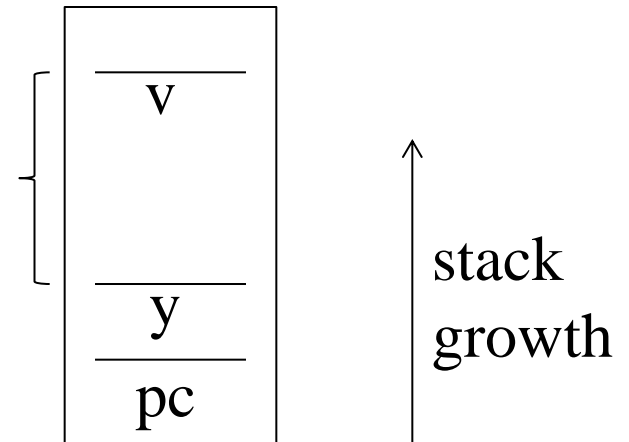
- The location targeted by the memory overwrite operation is located in the current stack area
- As the bare minimal, this is the location that contains the **return address** of the currently executed machine routine
- So, if the machine routine shuts down its stack frame and then returns, control can reach any point in the address space

A scheme

✓ when a call to a procedure is executed the following steps take place:

1. Parameters might be copied into the stack
2. The PC return value is then logged into the stack
3. Stack room is reserved for local variables

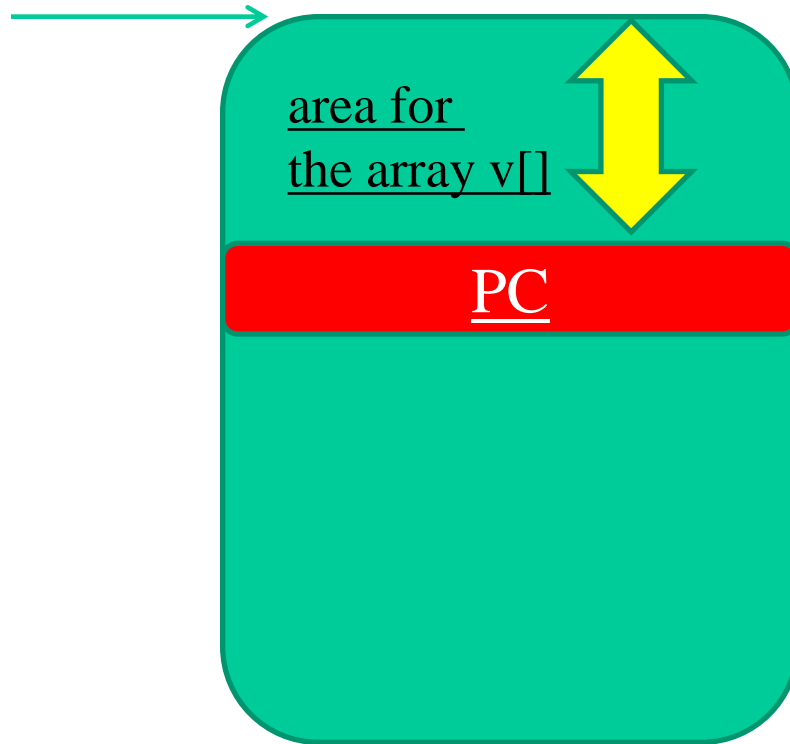
```
void do_work(int x) {  
    char v[SIZE];  
    int y;  
    .....  
}
```



- The `v` buffer could be used with no explicit control on its boundaries, this may happen when using classical standard functions like `scanf/gets`
- **This may also occur because of a bug on pointers handling**
- This limitation can be exploited in order to impose a variation of the control flow by overwriting PC
- This is also called **stack exploit**
- Control can be returned either to the original code or to a new injected one
- If the target code is injected, we say that the attack is based on external job – **stack exploit with payload**

E baseline example of buffer overflow

stack pointer
as seen by `f()`



Stack area

```
void f() {  
    char v[128];  
    .....  
    scanf ("%s", v);  
    .....  
}
```

Strings longer than
128 will overflow
the buffer `v[]`

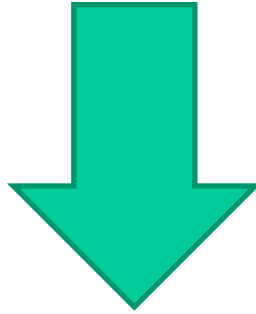
Risk of destroying
PC value

Examples of deprecated functions

`scanf()`

`gets()`

Libraries typically make available variants where parameters allow full control in the boundaries of memory buffers

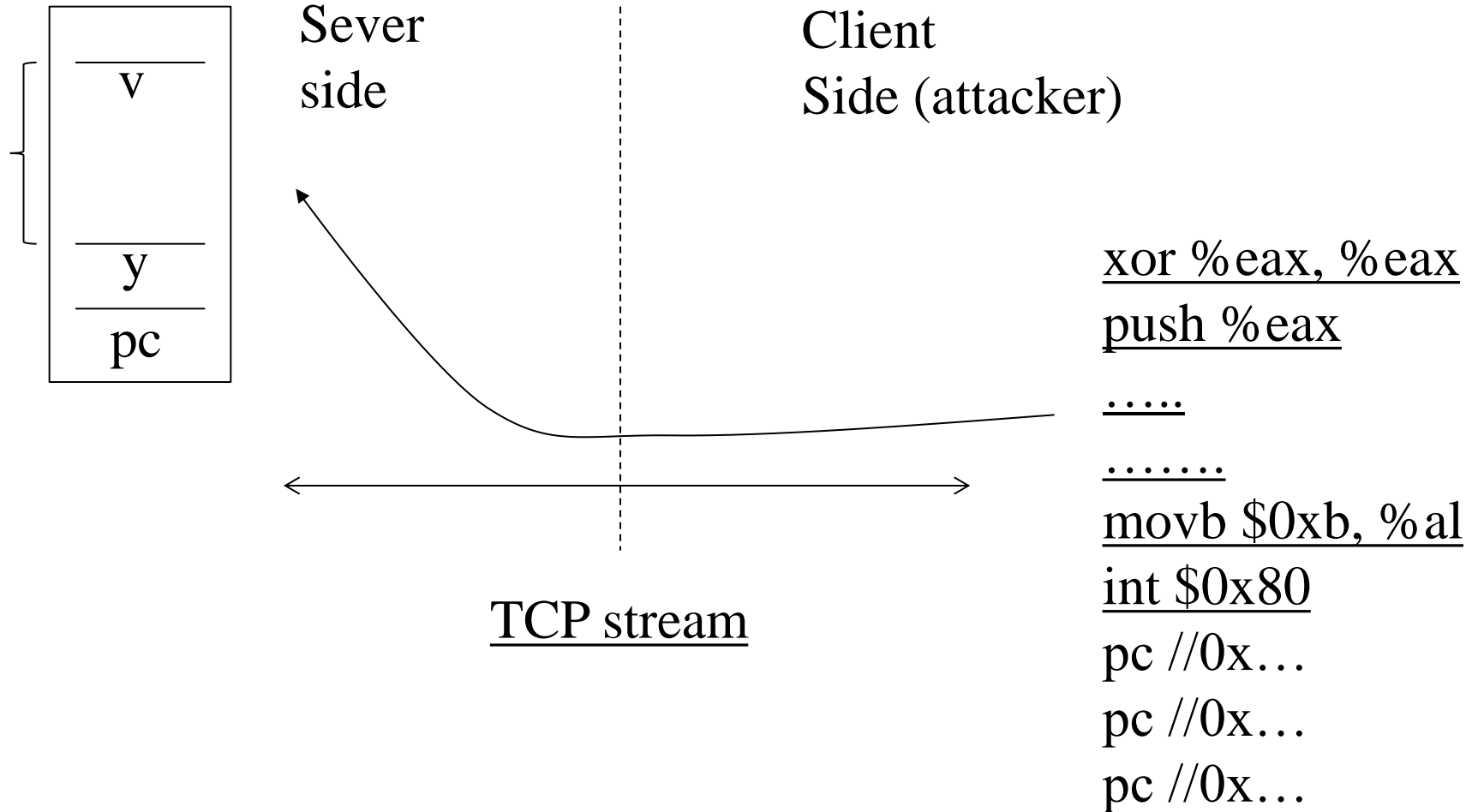


`scanf_s()`

Important notice

- Buffer overflows may also be linked to simple software bugs
- We may have bad usage of pointers, so that even if we use non-deprecated functions, we may still pass some wrong pointer leading to overwrite some memory location in a software unsafe manner

Another example scheme



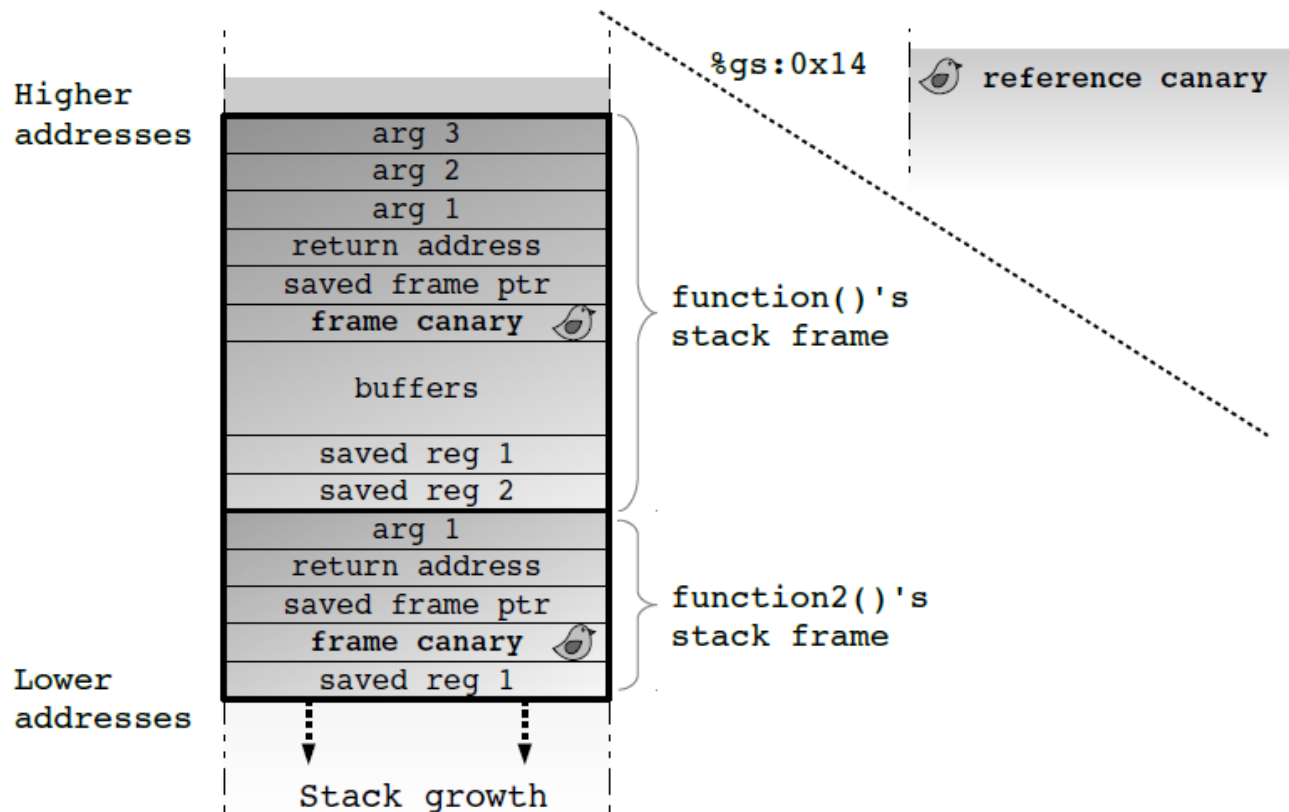
On improving the attack success probability

```
nop  
nop  
nop  
.....  
nop  
nop  
nop  
xor %eax, %eax  
push %eax  
....  
.....  
movb $0xb, %al  
int $0x80  
pc //0x...  
pc //0x...  
pc //0x...
```

this widens the likelihood of actually grubbing control and can also reduce the number of tries (namely PC values to be tried)

Buffer overflow protection methods: the canary tag

- Canary random-tags as cross checks into the stack before exploiting the return point upon the `ret` instruction
- This is the (nowadays default) `-z stackprotector` option in `gcc`



Executable vs non-executable address space portions

- x86-64 processors provide page/region protection against instruction-fetches
- This is the XD flag within the entries of the page tables
- Such a support was not present in 32-bit versions of x86 machines
- This is one reason why the `PROT_READ/PROT_EXEC` flags of `mmap ()` are sometimes collapsed onto a same protection semantic
- To enable instruction-fetches from the stack on x86-64 you can use the “`-z execstack`” option of the `gcc` compiler

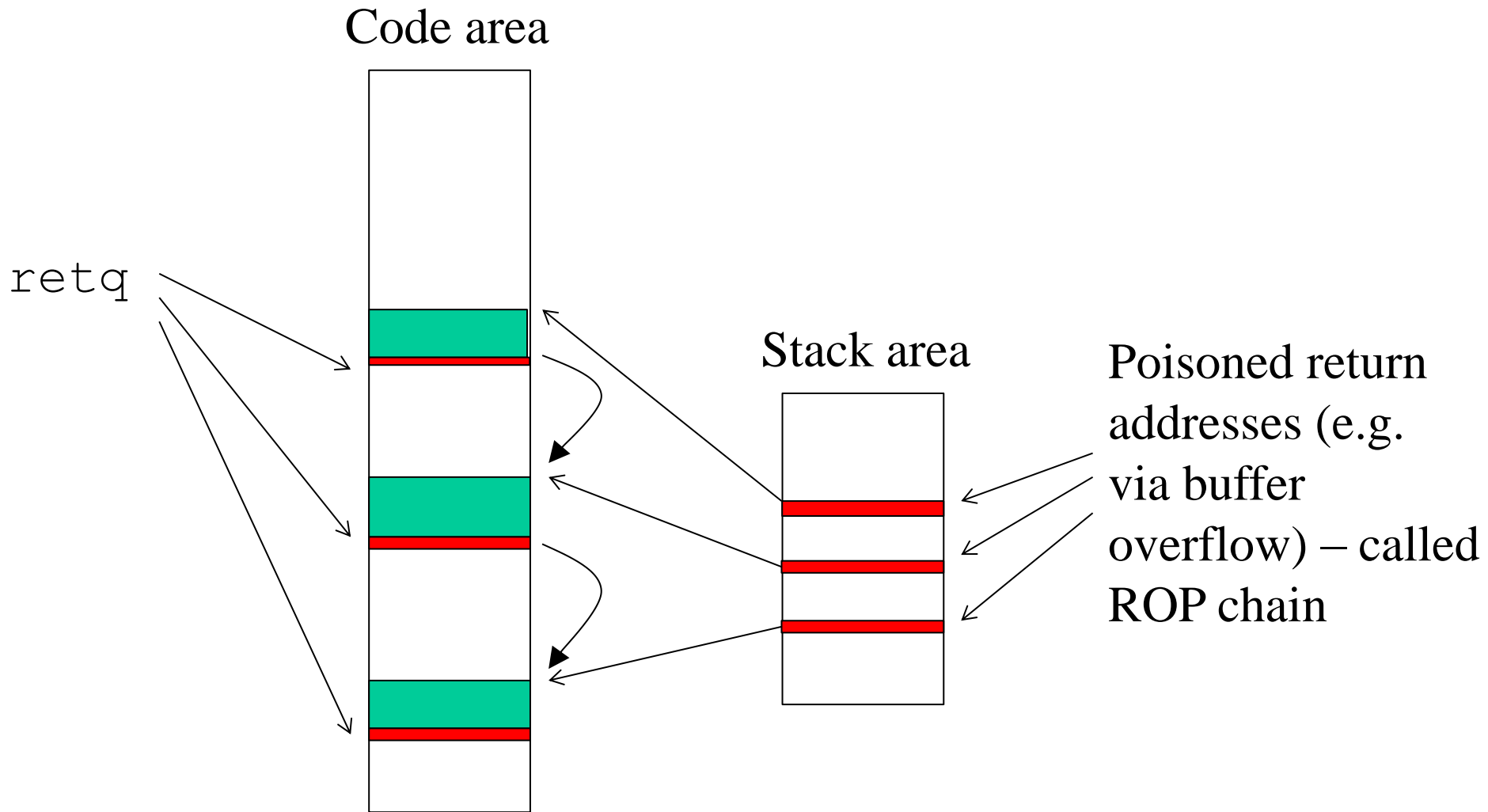
Are we finally safe??

- We cannot install code wherever we want, since flags like XD will not allow us to run whatever we would like from stack or data OS pages
- However, as we saw, running an exec for activating a new program is a matter of very few machine instructions
- These instructions could be already present into the executable the thread is running so
- Why not doing a patch work and using them all together even if they are scattered into the address space??

ROP (Return Oriented programming)

- Rather than using a single poisoned return address we use a set
- Each element in the set returns control to a code portion that will then return control to the subsequent element in the set
- It looks like we activated N calls to arbitrary pieces of code that in the end return control to each other
- These N pieces of code are typically named gadgets (a term we already saw while discussing of Spectre)

A ROP scheme



Countermeasures (so far)

- Run any function in a shadow stack area – requires compile time intervention – generates overhead
- Use the call/return hardware branch predictor to detect mismatches in between system calls
 - ✓ Does not cope with asynchronous flow control change
 - ✓ Requires serious patching of the system calls (via wrappers) to analyze the predictor state (via performance counters)

The actual damage by buffer overflows

- The buffer overflow attack can cause damages related to the level of privilege of the exploited application
- If the exploited application runs with SETUID-root then the attacker can even be able to get full control of the system, e.g. by manipulating the SETUID bit of the shell program
- actually the system root user is indirectly doing something non legitimate!!

User IDs in Unix

- The username is only a placeholder
- What discriminates which user is running a program is the UID
- The same is for GID
- Any process is at any time instant associated with three different UIDs/GIDs:
 - ✓ Real – this tells who you are
 - ✓ Effective – this tells what you can actually do
 - ✓ Saved – this tells who you can become again

UID/GID management system calls

- `setuid()` / `seteuid()` – these are open to UID/EUID equal to 0 (root)
- `getuid()` / `geteuid()` – these are queries available for all users
- similar services exist for managing GID
- `setuid` is “non reversible” in the value of the saved UID – it overwrites all the three used IDs
- `seteuid` is reversible and does not prevent the restore of a saved UID
- ... an EUID-root user can temporarily become different EUID user and then resume EUID-root identity
- UID and EUID values are not forced to correspond to those registered in the `/etc/passwd` file

An example

	UID	EUID	saved-UID	

seteuid	x	0	0	
	x	y	0	Line not flushed to x
setuid	x	x	0	since UID and EUID
	x	0	0	(or EUID/saved-UID)
setuid				are not the same

Operations by `su` / `sudo` commands

- Both these commands are `setuid-root`
- They enable starting with the `EUID-root` identity
- The subject to correct input `passwd` by the used, they move the real `UID` to `root` or the target used (in case of `su`)
- After moving the `UID` to `root`, `sudo` execs the target command

Coming back to non-legitimate code usage

- How to prevent that non-legitimate usage occurs along threads running on behalf of the root-user??
- This is a matter of making **the operational root of a system stand as something like a regular user**
- So who should really administrate security in our software system?

Secure (not only security enhanced) operating systems

- A secure operating system is different from a conventional one because of the different granularity according to which we can specify resource access rules
- This way, an attacker (even an actual user of the system) has lower possibility to make damages (e.g. in term of data access/manipulation) with respect to a conventional system
- Secure operating systems examples in the Linux world are:
 - SELINUX (by NSA)
 - SecurLinux (by HP)
- Secure operating systems rely on the notion of **protection domain**

Protection domain

DEFINITION: a protection domain is a set of tuples
 $\langle resource, access-mode \rangle$

- If some resource is not recorded in any tuple within the domain associated with users or programs then it cannot be accessed at all by that user/program
- Otherwise access is granted according to the access-mode specification
- The philosophy that stands beside operating systems relying on protection domains is the one of always granting the minimum privilege level

- Sometimes the protection domain is associated with individual processes (rather than users/programs)
- Therefore it can even be changed along time (generally by reducing the actual privileges)
- Hence different instances of the same program may have different protection domains associated with them
- So privilege reduction for a given process does not compromise correct functioning of other process instances

Advantages from protection domains

- Let's suppose an attacker grabs access to the system, e.g. via a bug that subverts authentication
- His potential for damage is bounded by the actual protection domain of the process that has been exploited in the attack
- As an example, if the attacker exploits the web server, the damages are bound by the protection domain of this server

Security policies

DEFINITION: a security policy is termed discretionary if ordinary users (including the administrator/root user) are involved in the definition of security attributes (e.g. protection domains)

DEFINITION: a security policy is termed mandatory if its logics and the actual definition of security attributes is demanded to a security policies' administrator (who is not an actual user/root of the system)

Security policies vs secure OS

- A secure operating system does not only require to implement protection domains, rather it also needs mandatory security policies
- In fact, if discretionary policies were used, then domains would have no actual usefulness
- Conventional operating systems do not offer mandatory policies, rather discretionary ones (such as the possibility to redefine file system access rules by the users, including root)

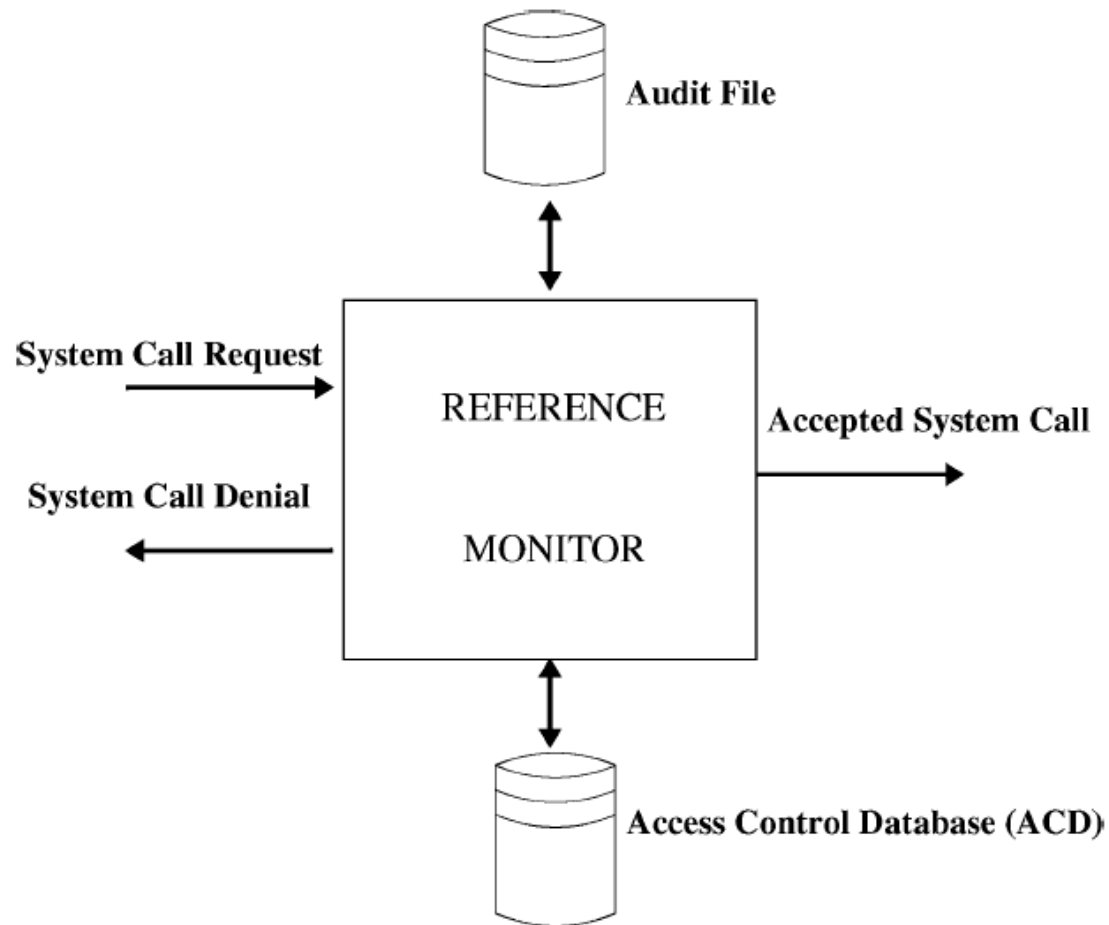
Secure operating systems administration

- In a conventional operating systems the root user is allowed to gain/grant access to any resource
- If an attacker grabs root permission then it can do whatever he would like
- In a secure operating system even root undergoes protection domain rules, as specified by the security administration, and as setup at system startup

Intrusion Prevention: Reference Monitors

- They aim at early detecting and stopping (the effects) of an attack
- Generally speaking they operate at kernel level, within secure operating systems
- The kernel modules forming the intrusion Prevention System are globally referred to as **reference monitor**
- Typically, these modules supervise the execution of individual system calls allowing the job to be carried out only in case parameters and system state match what is specified within an Access Control Database (which is based on **protection domains**)
- Close relation with the mandatory model

A classical Reference Monitor scheme



An example usage

- some SETUID application can be subject to a buffer overflow attack
- In case the application is not actually run by root, the dangerous system calls can be forbidden (such as the one that opens SETUID to programs)
- They can be done in real-time by the reference monitor on the basis of the ACL
- Particularly, the treatment of user ID and effective user ID in the context of buffer overflow can be based on the following macro applied to `current`

```
#define IS_SETUID_TO_ROOT(proc)  ! ((proc)->euid) && (proc->uid)
```

A second example

- We can discriminate whether specific services can be executed by root or SETUID processes depending on whether these are daemons or not (interactive ones)
- This can be still done in real-time by the reference monitor via the reliance on the ACL
- particularly, daemons targeted by buffer overflows can be treated by relying on the following macro applied to `current`

```
#define IS_A_ROOT_DAEMON(proc) !((proc)->euid) && ((proc)->tty==NULL)
```