Advanced Operating Systems (and System Security)
MS degree in Computer Engineering
University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Software security aspects

1. Recap
2. Classical pure-software vulnerabilities
3. Protection domains and secure operating systems
4. Reference Monitor architectures

IT security - the very baseline

1. Systems/applications <u>must be usable by **legitimate** users only</u>
2. Access is granted on the basis of an authorization, and according to the rules established by some <u>**administrator (beware this term)**</u>
   - ➢ 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 <u>trading-off operation acceptance (habilitation) vs current resource usage</u>

- 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 <u>**not exactly**</u> 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**
  - ➢ Data
  - ➢ Code or portions of it (either user or kernel)

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

Security approaches

- They are typically 3
  1. Cryptography (e.g. for data)
  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 data - 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
  - For streams
  - For device blocks
  - For memory pages/locations
  - For generic data (e.g. passwords)

This should come from other courses

Password cyphering

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

ID	Method
1	MD5
2a	Blowfish (on some Linux distributions)
5	SHA-256 (since glibc 2.7)
6	SHA-512 (since glibc 2.7)

Encryption library function

```c
#include <unistd.h>
char *crypt(const char *key, const char *settings)
```

The original passwd

Encryption algorithm (the method) + salt

Encryption method+salt+encrypted passwd

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: **<u>a piece of code that can be run along that thread</u>**
- 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 <u>make non-legitimate usage of memory locations, including blocks of code</u>
- 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 <u>compose them by fractions we take somewhere</u>

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 (coda/data in the system)

Lets' begin from the beginning

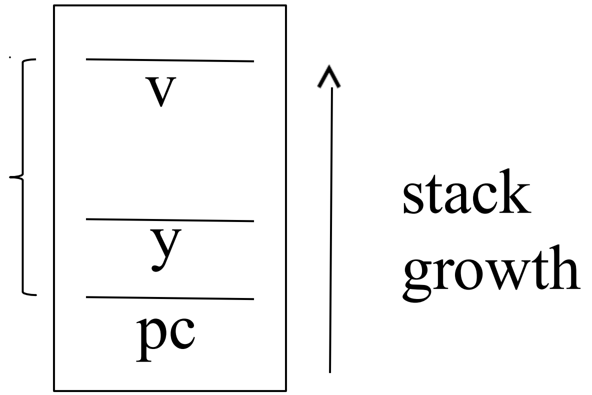
- The location targeted by the memory overwrite operation is located in the current stack area
- As the bare minimal, this is the location that contains the **return address** of the currently executed machine routine
- So, if the machine routine shuts down its stack frame and then returns, control can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

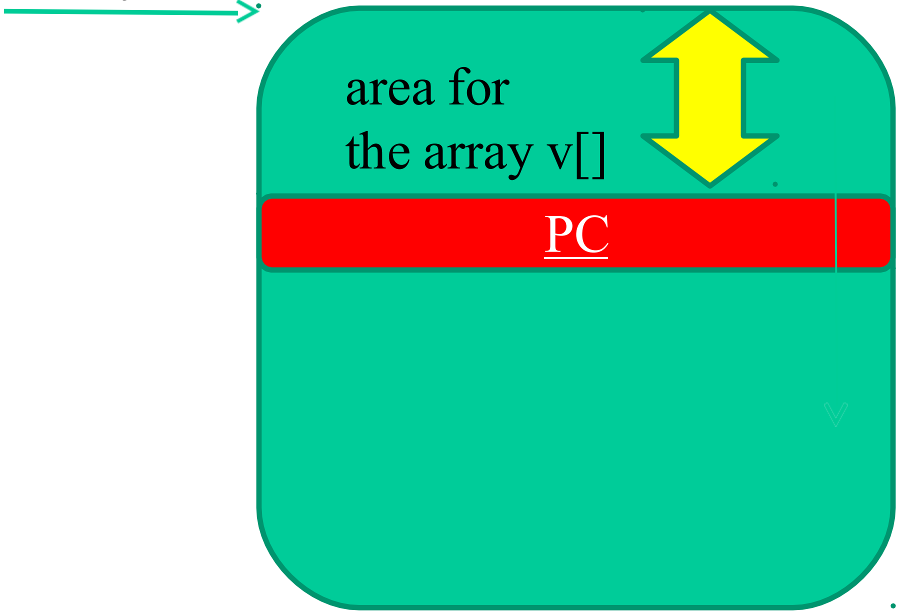
```
void do_work(int x){
   char v[SIZE];
   int y;
    ......

}
```



- The `v` buffer could be used with no explicit control on its boundaries, this may happen when using classical standard functions like `scanf/gets`
- <u>**This may also occur because of a bug on pointers handling**</u>
- This limitation can be exploited in order to impose a variation of the control flow by overwriting PC
- This is also called <u>**stack exploit**</u>
- 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 – <u>**stack exploit with payload**</u>

A 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()`

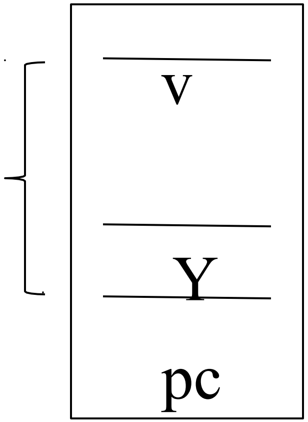
Libraries typically make available variants where parameters allow full control in the boundaries of memory buffers

`scanf_s ()`

Important notice

- Buffer overflows may also be linked to simple **software bugs**
- We may have bad usage of pointers
- Hence 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



<u>Server side</u>

<u>Client Side (attacker)</u>

TCP stream

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

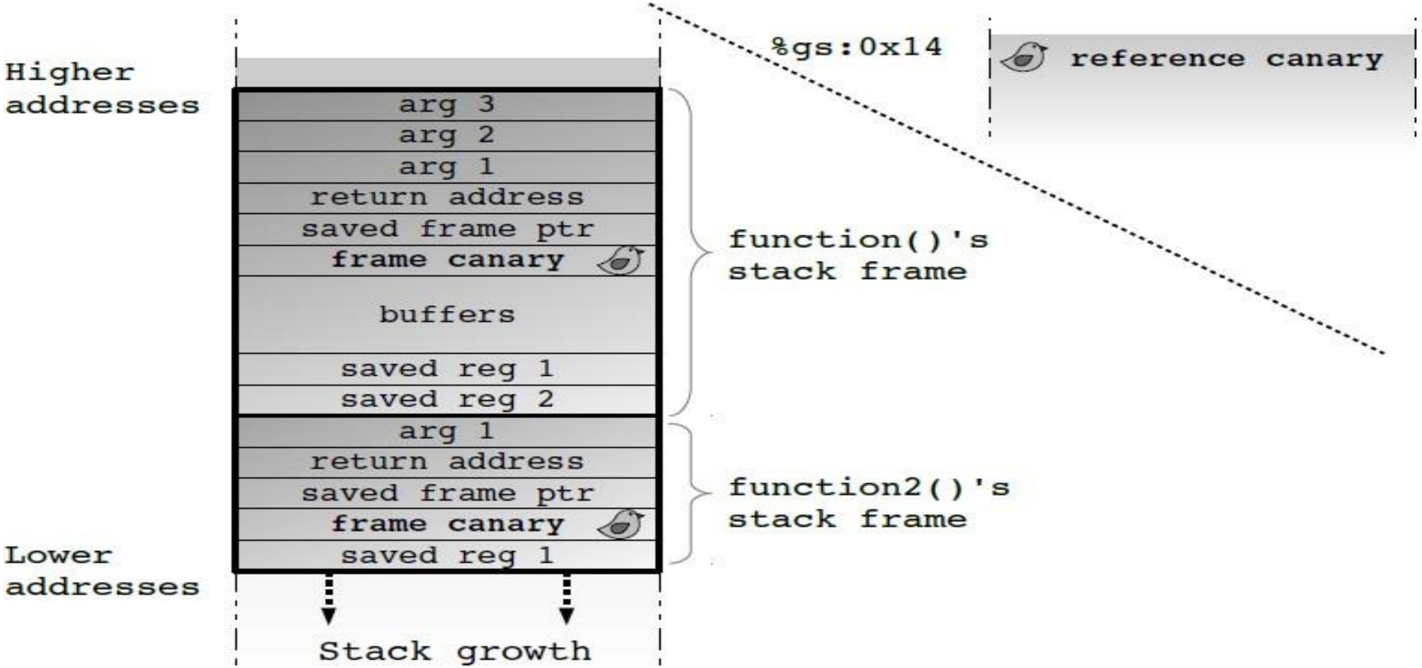
On improving the attack success probability

```
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

- To enable instruction-fetches from the stack on x86-64 you can use the "–z execstack" option of the `gcc` compiler

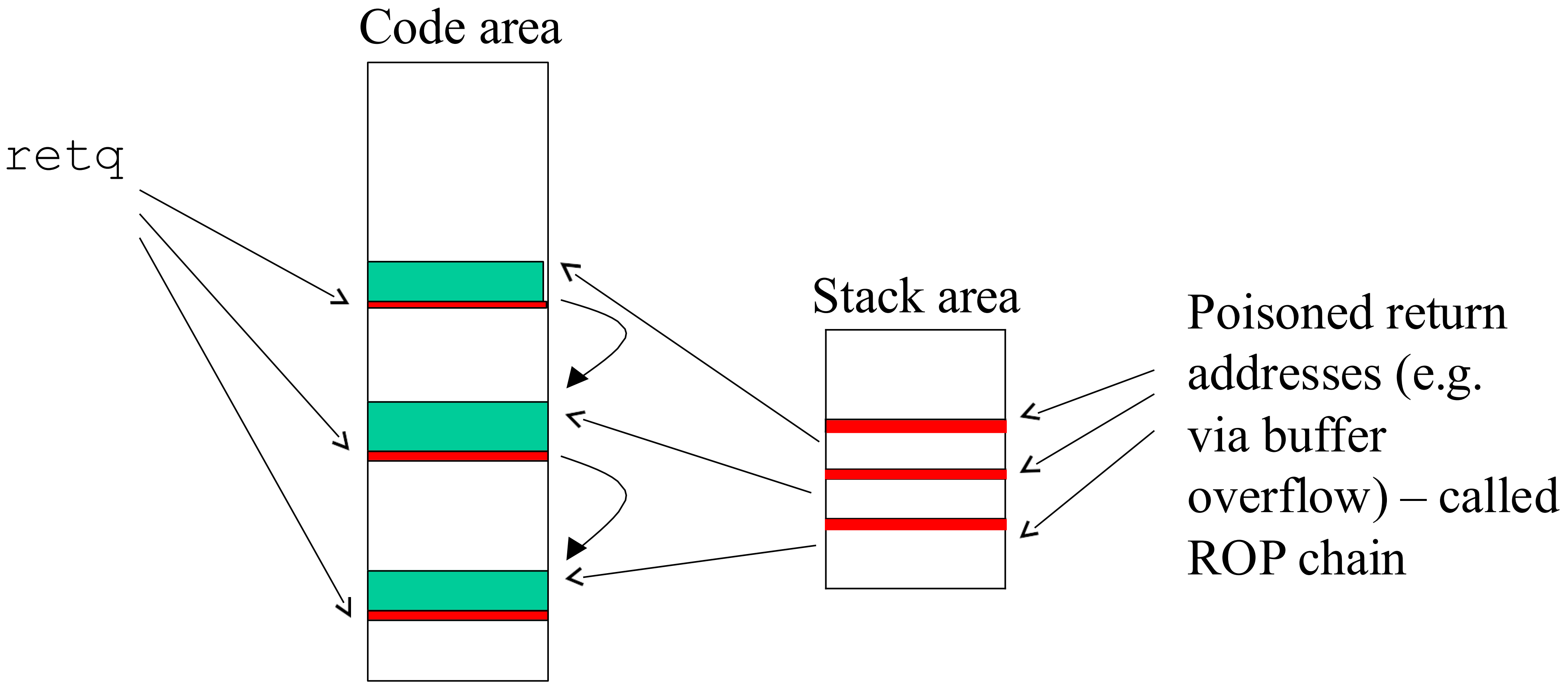
Are we finally safe?

- 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 cold 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


Code area
retq
Stack area
Poisoned return addresses (e.g. via buffer overflow) – called ROP chain

On the power of x86(-64) gadgets

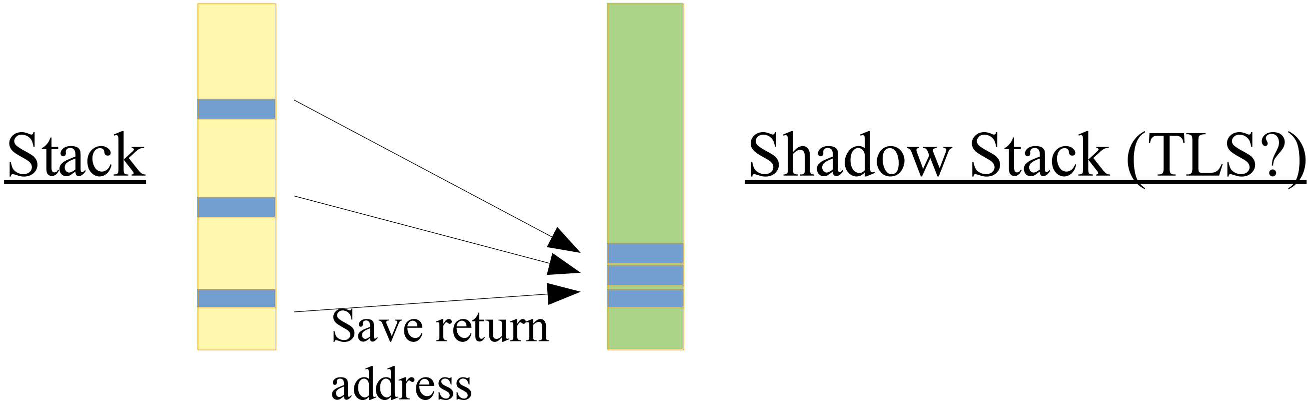
- x86(-64) does not impose alignment of instructions in memory
- A same code zone can be used in different manners via ROP
- As an example, for the code zone 55 48 89 e5 b8 00 00 00 00 5d c3 we have
  - From first byte <u>push %rbp   mov %rsp, %rbp   mov $0x0 %rax   pop %rbp   ret</u>
  - From third byte <u>in $0xb8,%eax   add %al,(%rax)   add %al,(%rax)   pop %rbp   ret</u>

Return Address Protection (RAP) – x86 case

- Each function is compiled to have <u>a preamble and a tail</u>
- In the preamble the return address is saved in an encrypted form into RBX, which is prior saved into the stack
- The encryption key can be the <u>current value of one register not used in the function if any</u>
- Before returning, the encrypted value in RBX is decrypted and compared with the return value, if the two are equal then we can actually return
- Still subject to attacker reads of the encrypted value and of the key if they must in turn be saved in memory (e.g. under high pressure on registers)
- It can have a non-minimal cost, especially for short living functions

Shadow stacks

- Each function is compiled to have a preamble and a tail
- In the preamble the return value is saved into a shadow stack area
- In the tail the saved return value is installed onto the actual stack
- Still subject to tampering of the shadow stack area
- Still has a cost in terms of machine cycles to be executed

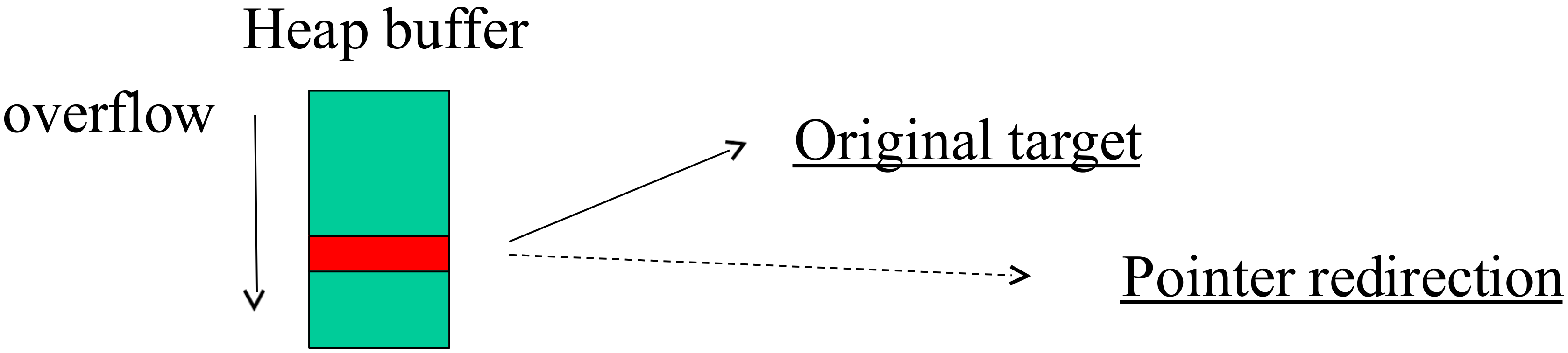


Other countermeasures (so far)

- Use the call/return hardware branch predictor to detect mismatches in between system calls
  - ✓ Does not cope with asynchronous control flow change
  - ✓ Requires serious patching of the functions/system-calls (via wrappers) to analyze the predictor state (via performance counters)
  - ✓ Still subject to excessive cost
- Exploitation of hardware-managed shadow stacks (see, e.g., CET)
  - It relies on the –fcf-protection flag of gcc
  - Full support is still far from implementation

Heap overflow

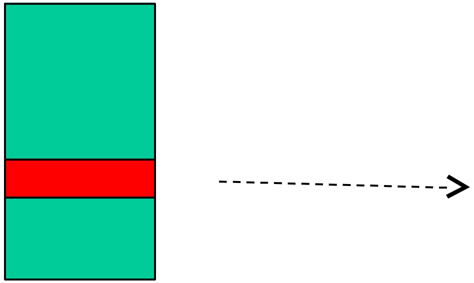
- It is an alternative way of attacking the memory layout of a running program (still because of an overflow)
- The target can be a function pointer, which can be redirected to already existing code or freshly injected one
- We can directly point to code to exploit data put onto the stack



Checking with the class of the target

- A possible way to protect against function pointer tampering is the check of the function pointer value before calling the target
- If the value belongs to a class of legitimate values, then the call can be executed
- Clearly we have a limitation of the possibility to exploit the funciton pointer to really point to arbitrary code zones
- We can also have a very large cost for each function pointer usage

Heap buffer



if pointer target not in the set S
of legitimate values (the correct class)
the call is aborted

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 (e.g. Linux)

- 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 (it is set to the effective value when acquired)

UID/GID management system calls

- `setuid()/seteuid()` – these are open to 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 UID-root user can temporarily become different EUID user and then resume UID-root identity
- UID and EUID values are not forced to correspond to those registered in the `/etc/passwd` file

An example

Non-privileged threads can only set to UID or saved-UID

```
seteuid(y)
setuid(x)
setuid(0)
```

UID	EUID	saved-UID
x	0	0
x	y	0
x	x	0
x	0	0

Line not flushed to x since EUID is not root

`setuid` changes all the three if the current euid is root (0)

Operations by su/sudo commands

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

Linux capabilities

- They allow the introduction of a **third** type of possibility to operate, which is between root and non-root

- Hence, if some thread needs to do something not allowed to non-root, it not necessarily needs to be a root thread

- Capabilities are also seen as an approach to build protection domains (<u>a thread has grants to do something but not everything</u>)

- To check the number of supported capabilities on Linux you can access the pseudofile /proc/sys/kernel/cap_last_cap

- To list the capabilities you can use “capsh --print”

A representation

Root thread	Non-root thread with capability	Non-root thread
(ID = 0)	(ID != 0)	(ID != 0)
All kernel level security checks are bypassed	Some kernel level security checks are bypassed	All kernel level security checks are executed

Capabilities masks

- a 32/64 bit mask is used to determine whether a thread has some capability
- several bit-masks are used to record
  - Permitted capabilities (what we can do)
  - Effective capabilities (the ones that we have now)
  - Inheritable capabilities (the once we leave to someone in exec)
  - Bounding capabilities (limit for inherit/permitted sets)
  - Ambient capabilities (what we allow to do with non-SUID programs, in any case limited by inheritable&permitted capabilities)

Exploitation

- Running as root allows all capabilities
- The `SECBIT_KEEP_CAPS` flag determines whether they are still kept when using `setuid()` under option PR_SET_SECUREBITS
- This flag can be configured based on the `prctl()` system call option

```
#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4, unsigned long arg5)
```

- After we change UID, we can release some capability

Linux system calls for process capabilities

```
#include <sys/capability.h>

int capget(cap_user_header_t hdrp, cap_user_data_t datap);

int capset(cap_user_header_t hdrp, const cap_user_data_t datap);
```

```
typedef struct __user_cap_header_struct {
        __u32 version;
        int pid;
} *cap_user_header_t;

typedef struct __user_cap_data_struct {
        __u32 effective;
        __u32 permitted;
        __u32 inheritable;
} *cap_user_data_t;
```

Listing process capabilities

- We can do this via /proc
- For each active process we have the /proc/PID/status pseudofile
- We can grep lines with the “Cap” string
- The outcoming bitmasks can be decode using the shell command
  capsh --decode=VALUE

File capabilities

```
#include <sys/capability.h>

cap_t cap_get_file(const char *path_p);

int cap_set_file(const char *path_p, cap_t cap_p);

cap_t cap_get_fd(int fd);

int cap_set_fd(int fd, cap_t caps);
```

Usable if the file system is mounted without the NOSUID option

Also. the user needs to have the CAP_SETFCAP capability available to set capabilities for files

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 **<u>the operational root of a system stand as something like a regular user</u>**
- 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
- SELinux (by the NSA) is an example of secure operating systems in the Linux world
- Secure operating systems rely (not only) on the notion of **<u>protection domain</u>**

Protection domain (i)

DEIFNITION: a protection domain is a set of tuples <resource, access-mode>

- If some resource is not recorded in any tuple within the domain associated with users or programs (**<u>or both</u>**) 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 **<u>minimum privilege level</u>**

Protection domain (ii)

- 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 grubs 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

Coming to the core - security policies

DEFINITION: a security policy is termed **discretionary** if ordinary users (including the administrator/root user) are involved in the definition of security attributed (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 (even for ACLs), 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 system the root user is allowed to gain/grant access to any resource
- If an attacker grubs 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

- compile/startup configuration of domains or ACL
- run-time external (policy server) reconfiguration for domains or ACL
- no operations allowed in site

security administrator

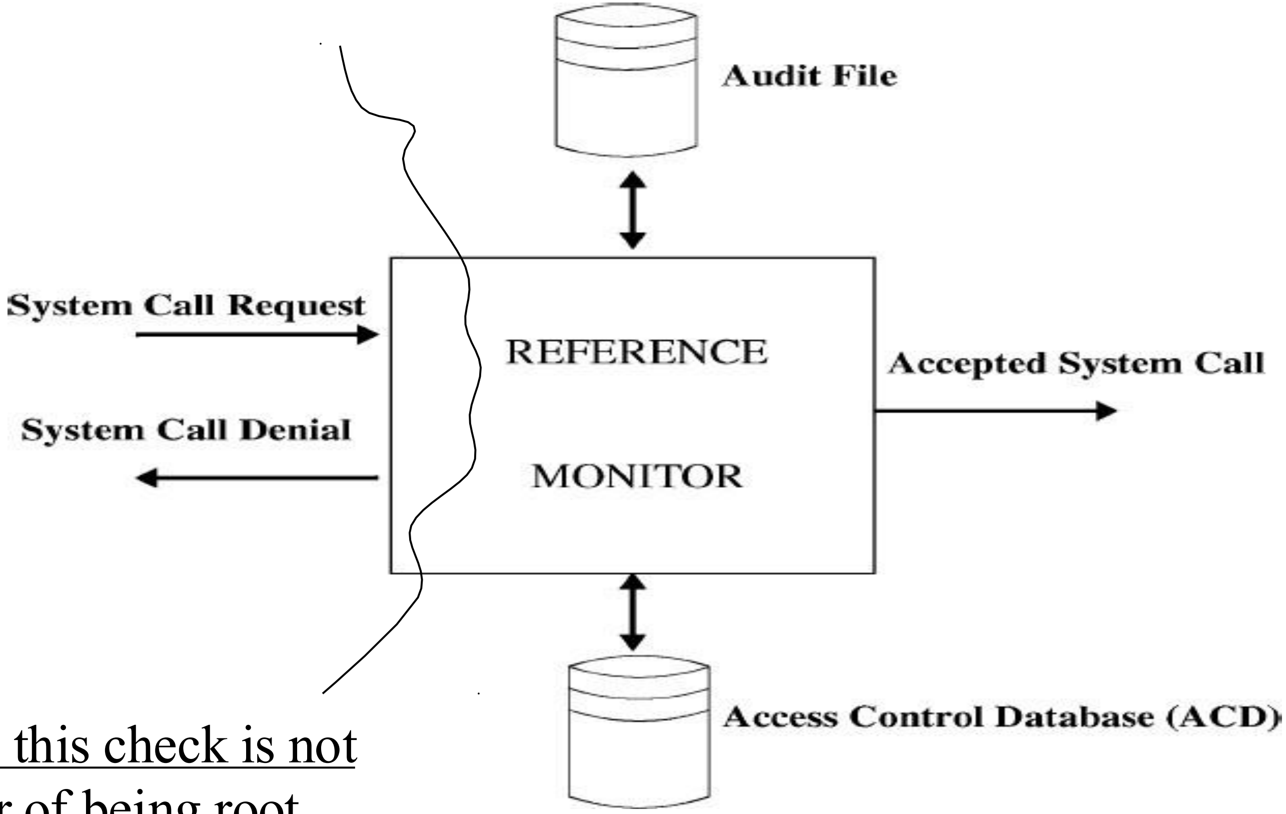
root user

- in site operations limited by domain or ACL

Reference monitors

- They aim at enforcing protection domains for any user, even
- Generally speaking they operate at kernel level, within secure operating systems (but we may have reference monitors for other layers such as databases)
- Typically, these modules supervise the execution of individual system calls allowing the job to be carried out only if parameters and system state match what is specified within an Access Control Database (which is based on **<u>protection domains</u>**)
- Close relation with the mandatory model

A classical reference monitor architecture



Passing this check is not a matter of being root

A usage example

- Some SETUID application can be subject to a buffer overflow attack
- If 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 its ACL
- Particularly, the treatment of user ID and effective user ID in the context of buffer overflow can be based on **detecting their values starting from `current`**

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 **discovering starting from `current` whether they have a valid terminal**