[Th 1] Control hijacking (dirottamento del controllo)

BUFFER OVERFLOW

Policy: The adversary can only perform operations intended by programmers

Threat model: the adversary can connect to the server and supply an input

The buffer overflow is an error in the code that can be exploited by hackers to gain unauthorized access to the system.
This error is based on buffers which are sequential sections of computing memory that hold memory as it is transferred between locations.
It occurs when the amount of data in the buffer exceeds its storage capacity. So this extra data overflows into adjacent memory locations and corrupts or overwrites the data in these locations.

Typically a buffer overflow attack involves violating programming languages and overwriting the bounds of the buffer they exists on.
It occurs typically when the code:

- depends on data given in input from the user
- depends on data that are enforced beyond its immediate scope

There are different types of buffer overflow that can be used:

- **stack-based buffer overflow:**
  - this is the most common and occurs when the attacker sends malicious code to an application that stores this data in the stack buffer. This overwrites the data in the stack, including its return pointer.
- **heap-based buffer overflow:**
  - this is more difficult because it needs to insert the attacker program memory beyond the memory used for the current runtime operations.
- **format string attack:**

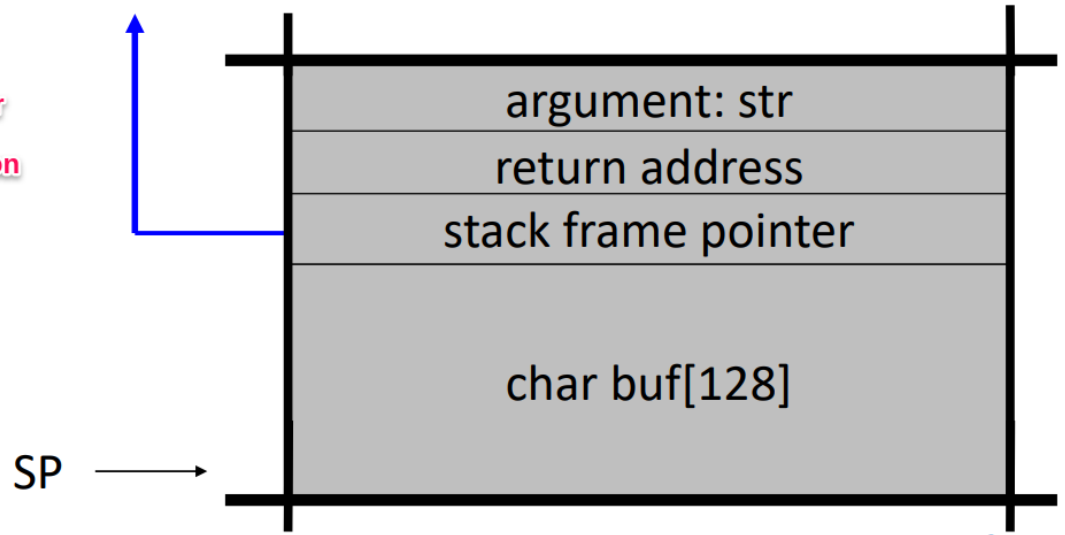
- it occurs when the application processes input data as a command or does not validate input data effectively. This enables the attacker to execute code, read data in the stack, or cause segmentation faults in the application.

EXAMPLE OF BUFFER OVERFLOW

```
void func(char *str) {
     char buf[128];
     strcpy(buf, str);
     dosomething(buf);
}
```

the size of the buffer is 128 bytes but there are no check on str

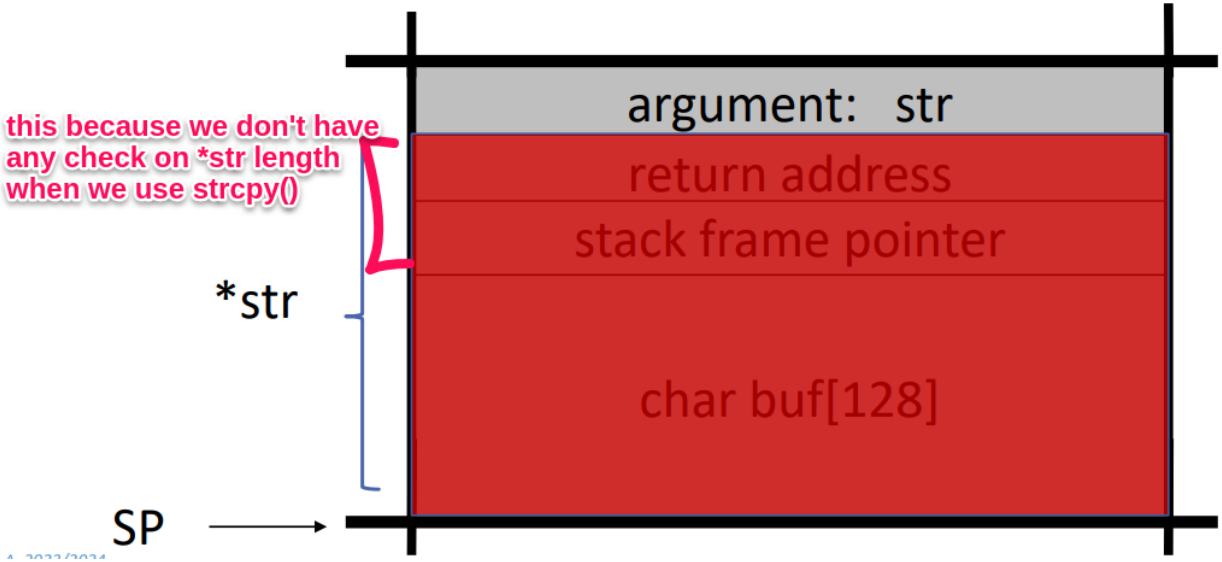
When func() is called, stack looks like:



Now let's imagine that an attacker inserts a string of 136 bytes instead of 128:

What if `*str` is 136 bytes long?

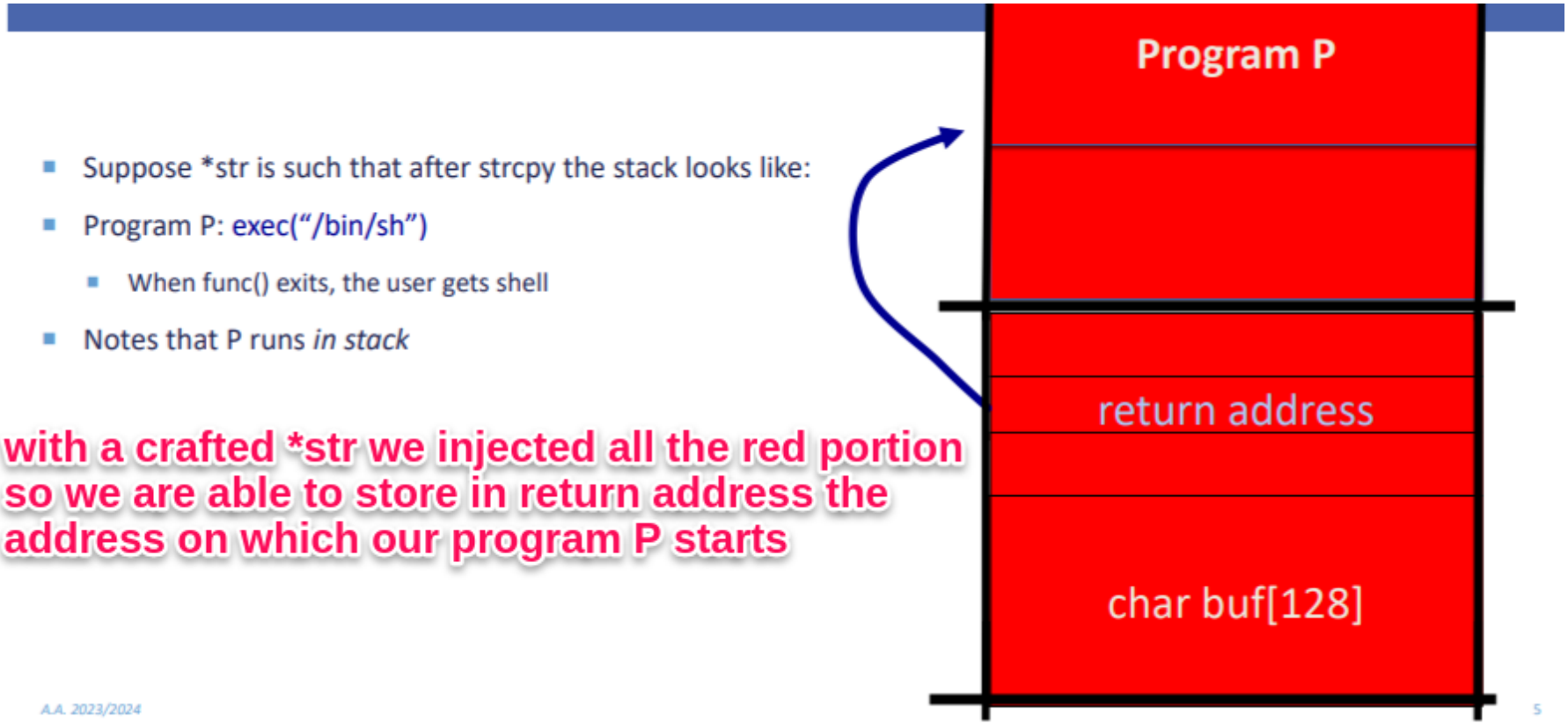
Problem: no length checking in strcpy()



Now we know that:

- **the stack frame pointer is 4 bytes for 32-bit processors or 8 for 64-bit processors**
- **the return address is 4 bytes for 32-bit processors or 8 for 64-bit processors**

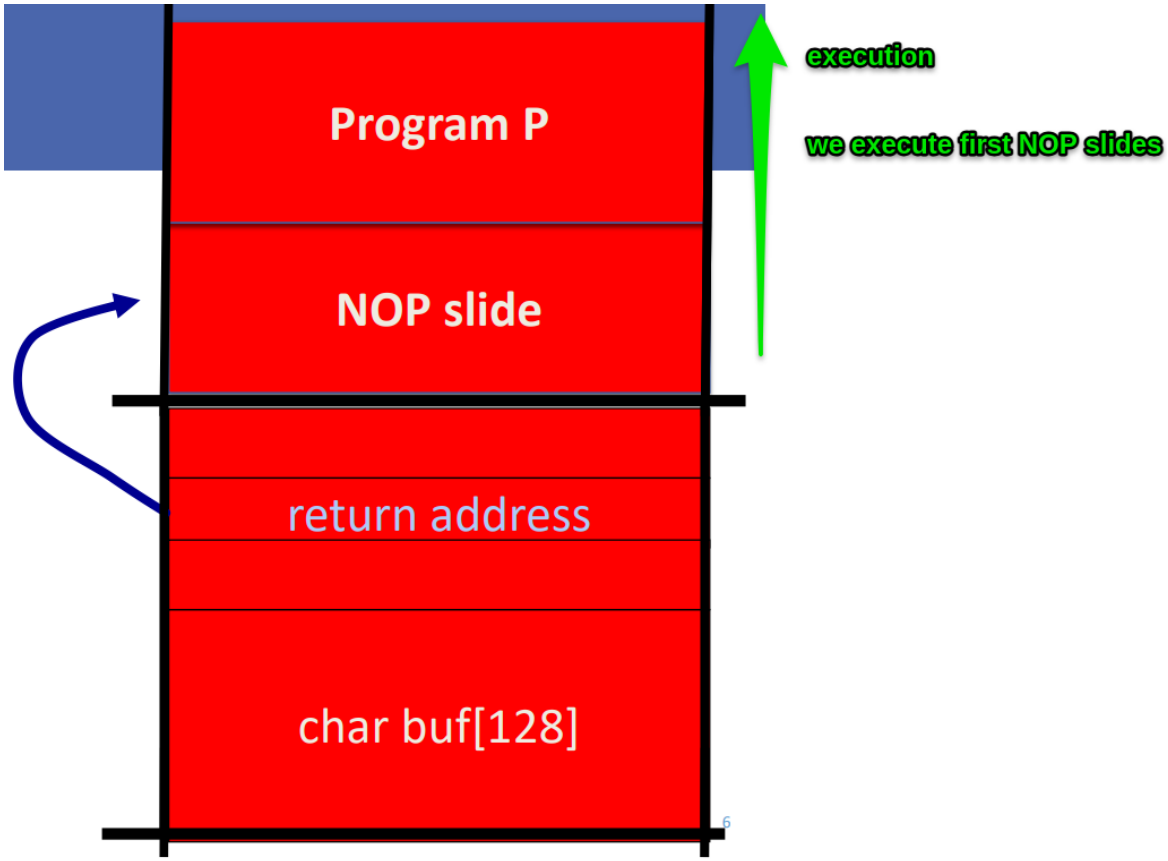
So we can craft a malicious *str in order to inject a malicious code in the stack :



How the attacker can determine the return address?

One solution is to use the NOP slide.

What we need to do is to force our code execution so we add NOP slides before of the code in order to force the return address to point on these NOP slides. So the program will execute the code from the NOP slide and after that will start to execute our code.



In general, we can guess the approximate stack state when the function fun() is called (in this case)

- virtual memory can make it easy

We can insert many NOP slides before the program P
Or we can increase the size of the target area.

It affects generally low-level languages (C and C++ are very diffused in the world, used for example for kernels, SQL servers, industrial systems, etc.) with relevant security implications.

Buffer overflow leverages on different facts:

- **legacy code not exposed to the internet**
- **C system software for velocity reason**
  - raw pointer exposed
  - no bounds checking
- **knowledge of the x86 architecture**
  - **structure of stack and heap**
  - **function call conventions**

Nowadays is difficult to use this type of overflow because the stack memory is randomized by the OS so is not possible to determine the structure of the stack

- **it must be randomized on each reboot**

Why didn't the OS notice this buffer overflow?

1. for the OS nothing strange happens
2. the OS protects only hardware tables to prevent a process from having access to tables not permitted (used by other processes)
   1. other this OS let's program to execute without problems
3. in addition, the OS table protection does not prevent the buffer overflow launched by a process against itself
   1. this is because the overflown buffer and the return address are in the process of valid address space

Fixing buffer overflow

First approach

First of all, we need to **avoid bugs, so a good thing is to** avoid problems by construction but it is difficult to ensure that code is bug-free.

Second approach

Use and build tools that help programmers to find bugs like program analysis.
Use it to find problems in source code before it's compiled.
Or test a large number of random inputs (FUZZING)

Example

```
void foo(int *p) {
 int offset;                    // not initialized
 int *z = p + offset;
 if(offset > 7)                 // is called only when is satisfied
     bar(offset);
}
```

Third approach

Use languages memory-safe languages like Java, C# or Python.

But we have limitations:

- some organizations have legacy code
- the programmers could need low-level access to HW
- we could need performance
    - but this was a huge problem in the past, now the performance depends generally on IO-bound

BUFFER OVERFLOW MITIGATION: STACK CANARIES

One of the oldest ones.
The idea is to understand if there was a buffer overflow.

We place a value between the buffer and the stack frame pointer that is called CANARY.
The attacker will overwrite everything and for this reason, will overwrite also the CANARY. So what we do is to check the canary before return.

- **if the attacker knows the canary he can resolve his problem**
- **so it must be a random value**

The canary must be:

- hard to guess
- resilient after overflow (resiste all'overflow)
  - example terminator canary that is "0.CR.LF.-1"
  - these chars are terminators of many vulnerable function (so if the attacker will use them then the overflow will stop)

Stack canaries don't catch overwritten pointers if they are used before returning

```
int *ptr = ...;   // I create a pointer with a value
char buf[128];  //the buffer is in the stack now
gets(buf);  // it is vulnerable so the attacker uses it and ptr will be rewritten with the attacker value
*ptr=5;     //this value is written on a memory address under the control of the attacker
```

Other cases are:

- heap object overflows (if the attacker rewrites it we can't notice it)
- malloc/free overflows

If the attacker reads the canary he can use it to perform a buffer overflow without problems.

BUFFER OVERFLOW MITIGATION: Bounds checking

It is based on checking if the pointers are in a valid range.

It is a challenge because in C can be hard to differentiate between valid and invalid pointers.

For example in this code is difficult to establish pointer correctness:

```
union u {
 int i;
 struct s {
     int j;
     int k;
 };
};
int *p=&(u.s.k);
```

The general problem is that in C a pointer does not encode information about the intended usage semantics of that pointer.

Some tools just try to guess this semantic and instead they just enforce the memory bounds on heap objects and stack objects

The usage of bounds checking typically requires changes to the compiler and programs must be recompiled.

At a high level the goal is:

- for a pointer p2 derived by p1, p2 should only be dereferenced to access the valid memory region of p.

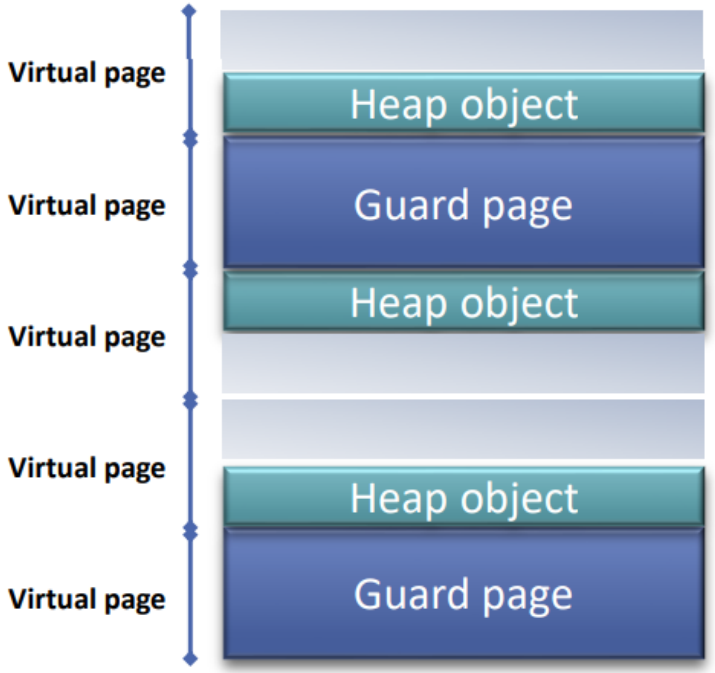
Bounds checking prevents arbitrary memory overwrites. A program can only use its memory only if it is actually allocated.

Approach 1 : electric fences

It is an old approach to protect the heap but simple.
Idea:

- **each one heap object is aligned with a guard page and we use page protection to ensure that accesses to the guard page cause a fault.**
- **a heap overflow/underflow will immediately cause a crash instead of silently corrupting the heap and causing a failure at some indeterminate time in the future**

It works without source code so we don't need to change compiler etc.
A bad thing is that it introduces a huge overhead, only one object in a single page and we need to overhead of a dummy page that isn't used for real data.



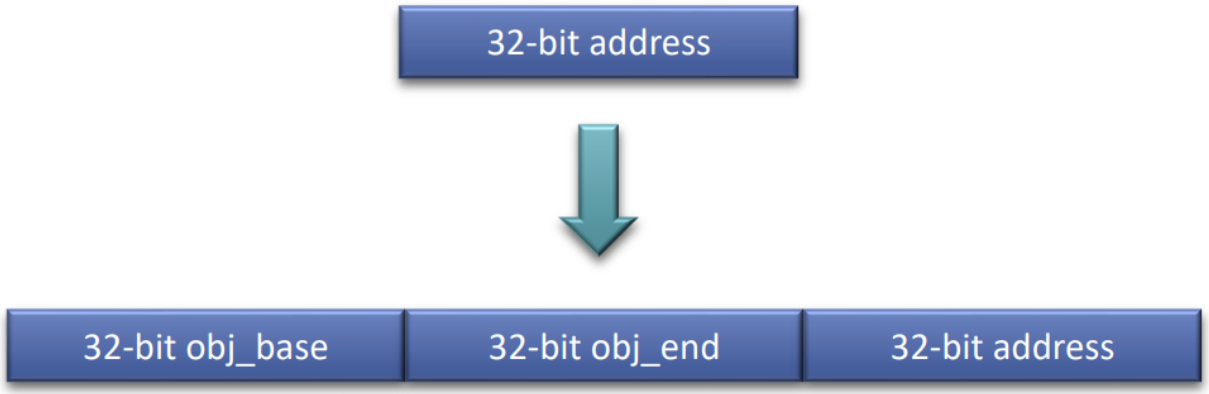
Approach 2: FAT pointers

We create a fat pointer where the original pointer is larger and contains:

- **a base pointer and an end pointer**
- **[base, end]**

The compiler generates an error code that aborts the program if in it we dereference a pointer whose address is outside of [base, end]

bad: But we convert a 32-bit pointer to 96-bit.
bad: They are not compatible with existing software and it can be expensive to check all pointer deferencies.



Approach 3: slow fat pointers

The idea is to store bounds information in unused pointer bits nd is used in 64-bit architecture.

Approach 4: shadow data structure

For each allocated object we store the size of this object in a data structure.

For each pointer we need to use two operations:

- pointer arithmetic, it is used to track the provenance of pointers and to tell when a derived pointer goes outside of the bounds of its base object
- pointer dereferencing, to take in count that an invalid pointer is not always a bug.

How we can store pointer information?

- using a hash table or interval tree to map addresses to bounds
  - space efficient
  - lookup can be slow
- using an array to store bounds info for every memory address
  - it is fast
  - high memory overhead

How we can force failing when we dereference out-of-bounds pointers?

- instrument every pointer deferences
  - expensive because we have to add extra code for each dereference

BAGGY BOUNDY TRICKS

there are 4 different main tricks

- **we round up each allocation to a power of 2**
- **we express each range limit as log_2(alloc_size)**
  - so basically each range has log2 bits to be represented with respect to object allocated size
  - **for 32-bit pointers we need 5 bits to represent all possible ranges**
  - **so the idea is that we can have pointers that cam move on these ranges, if a pointer changes the 6th bit then we are out of bounds**
- **we store limit info into a linear array**
  - fast lookup
  - virtual memory can be used to allocate the array on demand

- **we allocate memory at slot granularity (example 16 bytes)**
  - so we have fewer array entries

Example:
Given a pointer p1 and p2 derived from p1.
We can test if p2 is valid by checking whether pointers have the same prefix in their address bits (for 32-bit we need 5 bits) and by the fact that they can differ only on their least significant bits (in example the 5 bits).

```
                        slot_size = 16

p = malloc(16);         table[p/slot_size] = 4;

p2 = malloc(32);        table[p2/slot_size] = 5;
                        table[p2/slot_size + 1] = 5;
```

These buggy bounds can be applied to existing C systems but some legal C code that casts pointers to integers for example can be marked as illegal code. In addition could be different buffer overflows not catched, for example:

- arrays inside a malloc structure
- cast pointer to int and then to pointers again
- dangling pointer to reallocate memory

Costs of the bound checking

In general, we can have space overhead for storing bounds information, cpu overhead to execute pointers arithmetics and derefercing, false allarms

BUFFER OVERFLOW MITIGATION: NON EXECUTABLE MEMORY

Modern hardware allows you to put read, write, and execute permissions on memory. So for this reason we can mark the stack memory as non-executable in order to disallow the attacker to run its code.

In addition, some systems enforce a "W^X" policy that all memory is writeable or executable but not both

- **it can be neither of them**

Good: It potentially works without any application changes and the the hardware is watching you all of the time differently from the OS.
Bad: it is hard to generate dynamically the code

- java runtime and javascript engines generate x86 code on the fly

BUFFER OVERFLOW MITIGATION: Randomized memory addresses

This is used to make it difficult for the attacker to huess a valid code pointer

STACK RANDOMIZATION

The idea is to move the stack to random locations and/or place padding between stack variables.

This makes it difficult for the attacker to determine:

- where the return address of the current loaded frame is
- where the attacker code (shellcode) will be allocated

ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

The idea here is to randomize the entire address space (stack, heap, DLL locations etc)
In this case, dynamic loader can choose a random address for each library and program

```c
void main(int argc, char **argv) {

   int aVariable = 0;

   printf("Global is stored ad %p\n",&aVariable);

}
```

ASLR off

```
CC >./testaslr
Global is stored ad 0x7fffffffe4a4
CC >./testaslr
Global is stored ad 0x7fffffffe4a4
CC >./testaslr
Global is stored ad 0x7fffffffe4a4
CC >./testaslr
Global is stored ad 0x7fffffffe4a4
```

ASLR on

```
CC >./testaslr
Global is stored ad 0x7ffc1d5666f4
CC >./testaslr
Global is stored ad 0x7ffdd36e4bc4
CC >./testaslr
Global is stored ad 0x7fffa73e9a34
CC >./testaslr
Global is stored ad 0x7fffa8aa7e14
```

Is it possible to exploit random memory addresses?

The attacker can guess randomness:

- especially in 32-bit architecture because 12 bits are used to memory-mapped pages that need to be aligned to page boundaries (cannot be randomized)

- in 2004 we needed just 216 seconds to brute force it

An adversary might extract randomness
The attacker might not care exactly where to jump

- for example he will fill the memory with shellcode so that a random jump is OK

- Example: address leak in Flash's “Dictionary” (hash table)
  - Hash table internally computes hash value of keys
  - Hash value of an integer is the integer
  - Hash value of an object is its memory address
  - Iterating over a hash table is done from lowest hash value to highest hash value

1. Get victim to visit your Flash-enabled page
2. Create a Dictionary, insert a string object which has shellcode, and then insert a bunch of numbers into the Dictionary
3. By iterating through the Dictionary, determine where the string object lives by seeing which integers the object reference falls between
4. Overwrite a code pointer with the shellcode address and bypass ASLR!

CODE REUSE ATTACKS

Arbitrary code execution can be obtained through code reuse.

Code reuse attacks are software exploits in which the attacker directs control flow through existing code.

RETURN TO LIBC

Through a buffer overflow, the attacker replaces the return address with one of another function in the process
We can execute a shell with the same access rights as the executed program.

Example:

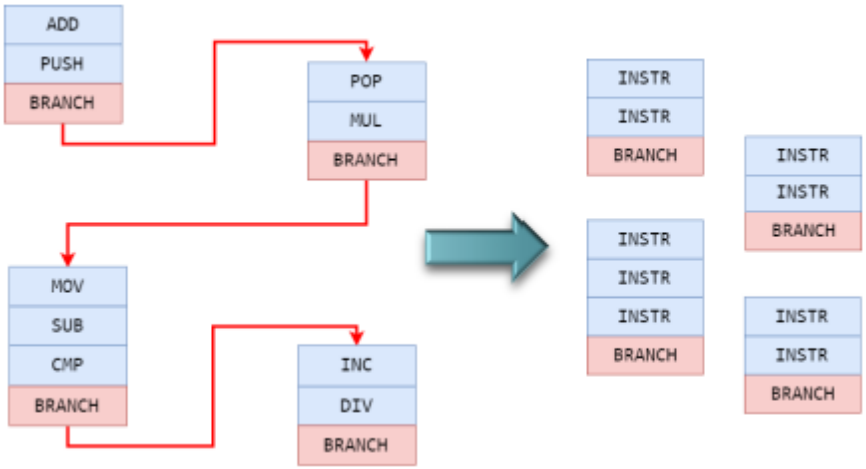
- we find the address of system() function
- we use "/bin/sh"

RETURN ORIENTED PROGRAMMING

In this case, **instead of using a single libc function to run our shellcode we concatenate together pieces of existing code that we call gadgets.**

The challenge is to find the gadget we really need to reuse and how to string them together.

Each **gadget is a instruction group that end with ret (typically of 2-5 instructions)**
They are invoked via ret instructions and can get theri arguments via pop, etc...

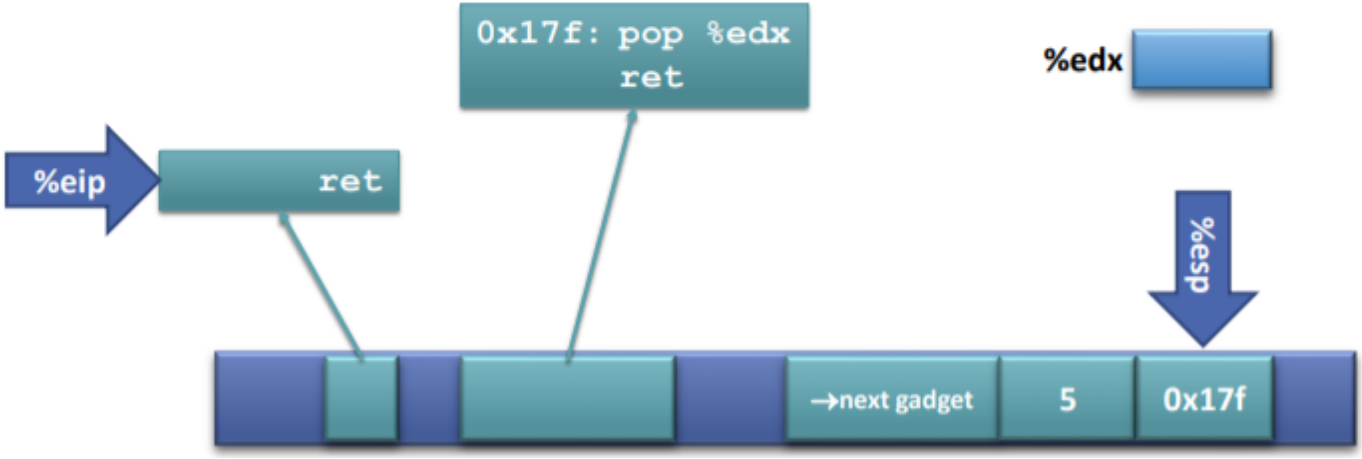


- Example: We want to move 5 into the %edx using a gadget

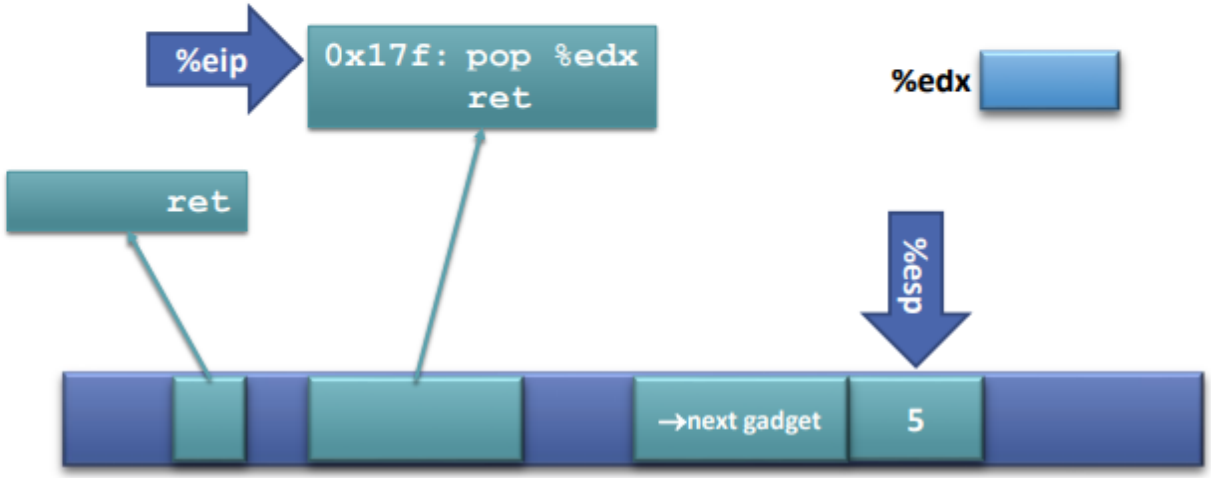
```
0x17f: pop %edx
       ret
```

- **ret** will pop off 0x17f

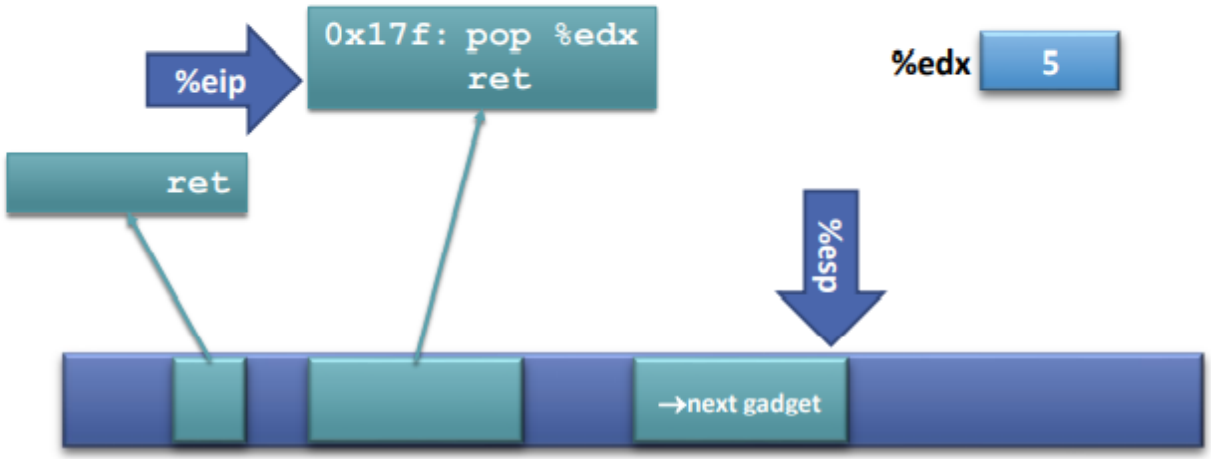
scopro che in 0x17f c'è questo gadget
che prende un valore dallo stack con pop
e lo mette in %edx



- The %esp points to 5
- The %eip points to the **pop** instruction
- The **pop** instruction will pop 5 off and put it into the %edx



- We have moved 5 into the %edx
- The %eip points to the **`ret`** instruction
- The %esp points to the start of the next gadget



We have 3 automated tools that can be used to find gadgets in memory:

- Ropper
- ROPGadget
- Pwntools

The defense from ROP is the use of ASLR (randomization of stack and heap)

But if a server restarts on a crash and it doesn't re-randomize then is possible to:

- read the stack to leak canaries and a return address
- find gadgets at run time that effect a call to the function write
- dumb binary to find gadgets for shellcode

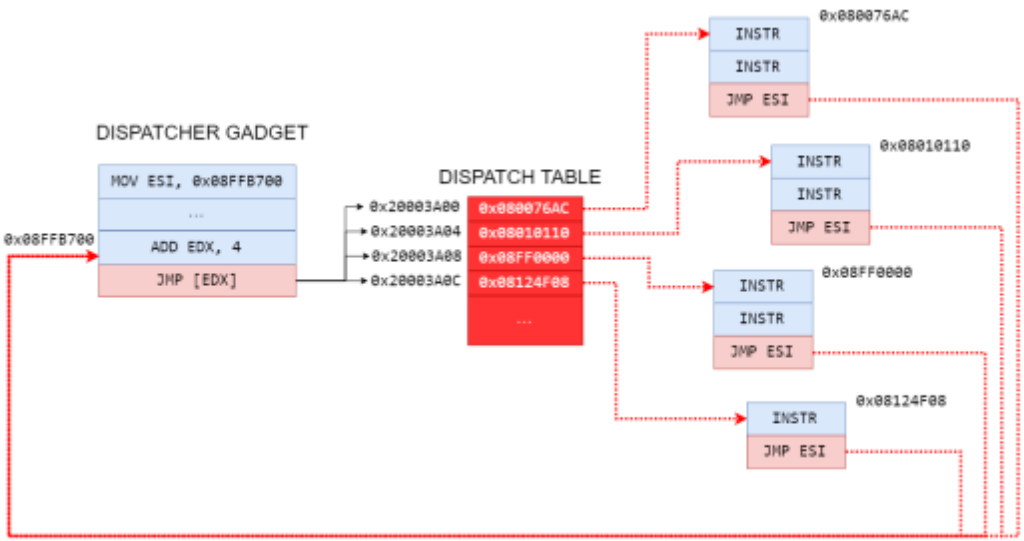
JUMP ORIENTED PROGRAMMING

The idea here is to trigger the execution of functions via a sequence of indirect jump instructions
Similarly to ROP the JOP is based on a sequence of small gadgets.

The idea is that each gadget is collected into a dispatch table that is located in a memory section afflicted by a vulnerability.

The final jmp instruction of the gadget is redirected to the dispatcher gadget that advances a pointer to the dispatcher table.

The dispatcher ends with an indirect jump to the address pointed by the pointer, in order to pass the control to the next gadget in the table.



CONTROL FLOW INTEGRITY (CFI)

Control flow integrity is the activity of checking that the program is behaving in the correct flow.

The challenge here is to define the expected behavior of the program and avoid that the detector program is compromised.

In general:

- the classic CFI (2005) imposes 16% of overhead or 45% in the worst case
- Modular CFI (2014) imposes 5% overhead on average and 12% in worst case and can eliminare 95,75% of ROP gadgets on x86-64

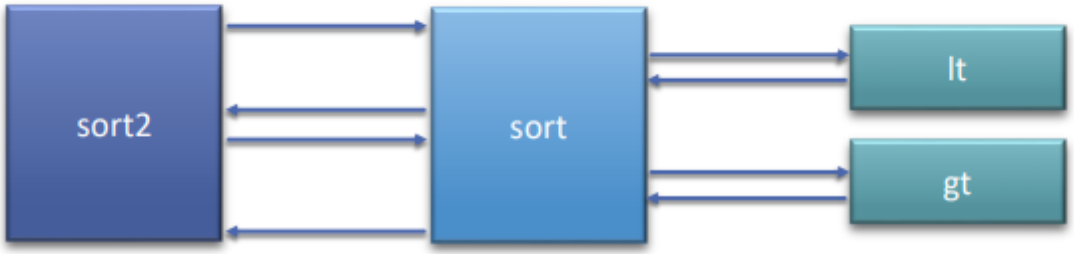
Control flow graph (CFG)

It is used to define the expected behavior.

Example:

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

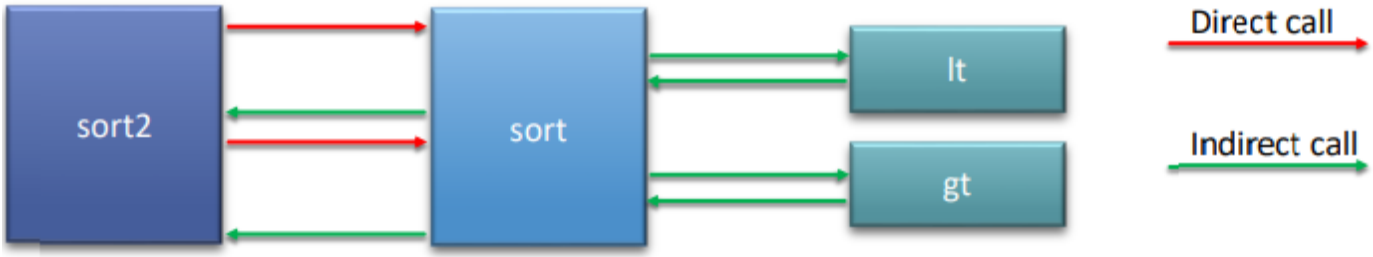
```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



The normal thing done by the compiler is that the function are compiled 1 time and for this reason:

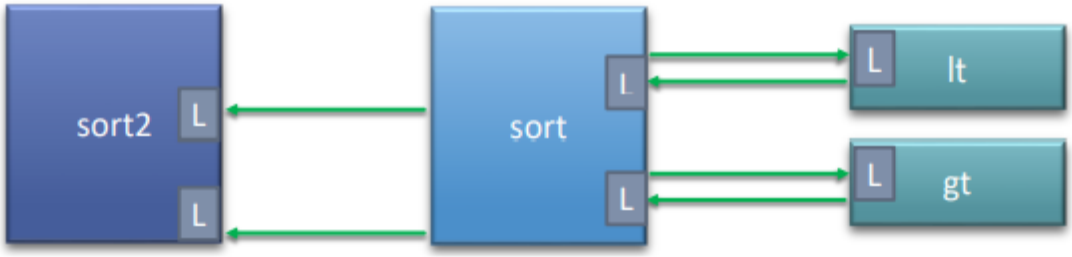
- sort2 function performs direct calls to sort

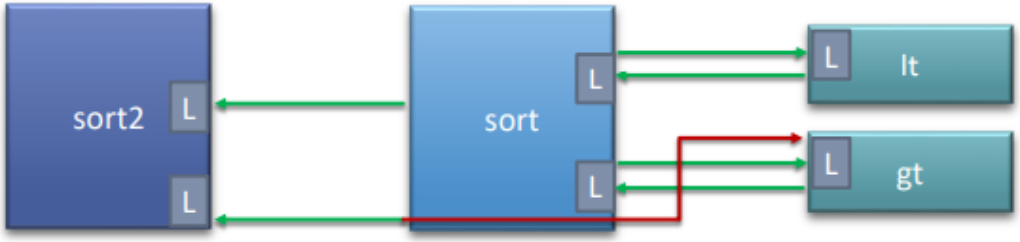
- but sort performs indirect calls to the function used as comparators.
- for this reason when sort returns to sort2 does not predict in which point it will return on run time
    - they are indirect calls that depends on the other sort execution



The idea to ensure that the program will follow the right flow is to insert labels just before the target address of an indirect transfer.
So we insert a code to check that the label of the current indirect transfer is the label of the normal flow function we need to call.
We abort if the labels does not match (an attacker has changed the call and redirect it to another function with another label or without a label at all).

Now, if we use the same label for each one function then we have a problem because the attacker make the program to jump to another function that is labeled with the same label as the others.

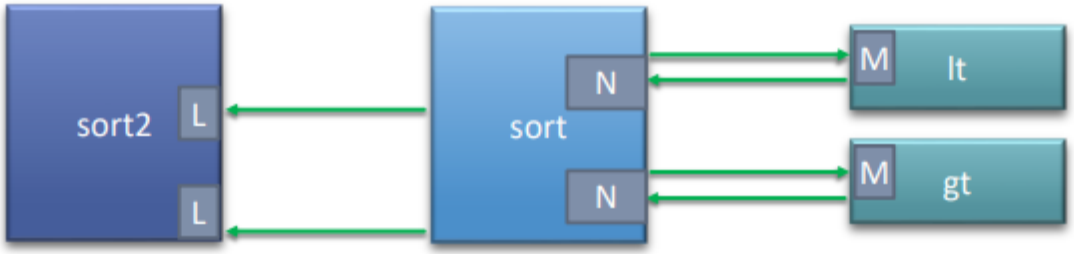


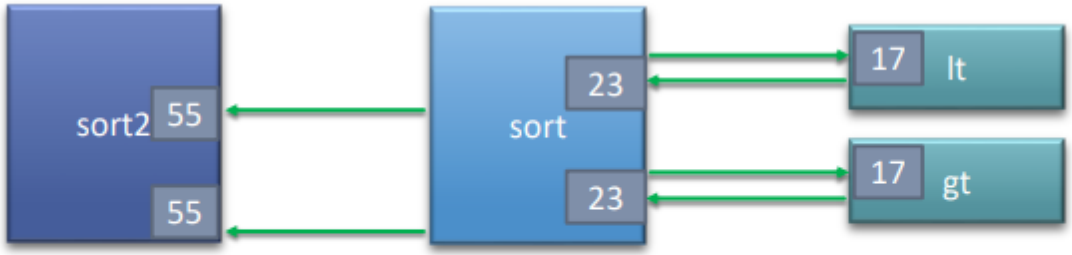


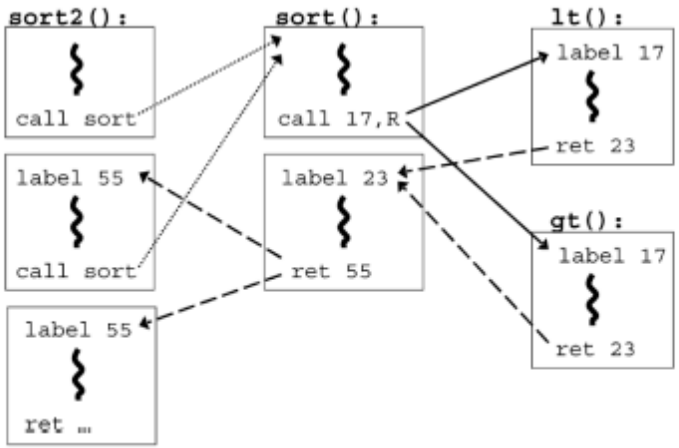
IN-LINE REFERENCE MONITOR

So basically we need to assign labels in a way that the normal flow can be ensured, so at compiling time we have to assign labels in a way that is not possible to make strange jumps to other functions, in the previous example we need that:

- the return point of sort to sort2 must be labeled with a specific label (L) in order to force sort to return only on sort2 function
- the return of lt and gt must be labeled both with N because sort expects to obtain a comparator by its implementation, so when we compile it we don't know which comparator will be called. In these case we allow only lt and gt so with the label N is possible to force them to return only on sort function
- at the same time the call from sort to lt and gt must be labeled with M because we want to check that sort will call only lt or gt.







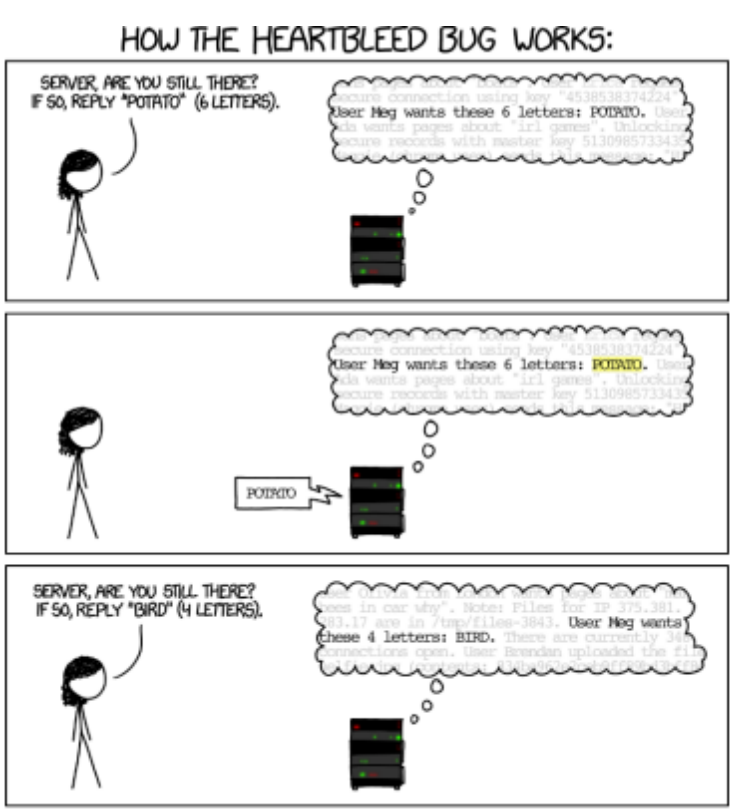
CFI defeats control floe-modifying attacks as remote code injection, ROP etc.

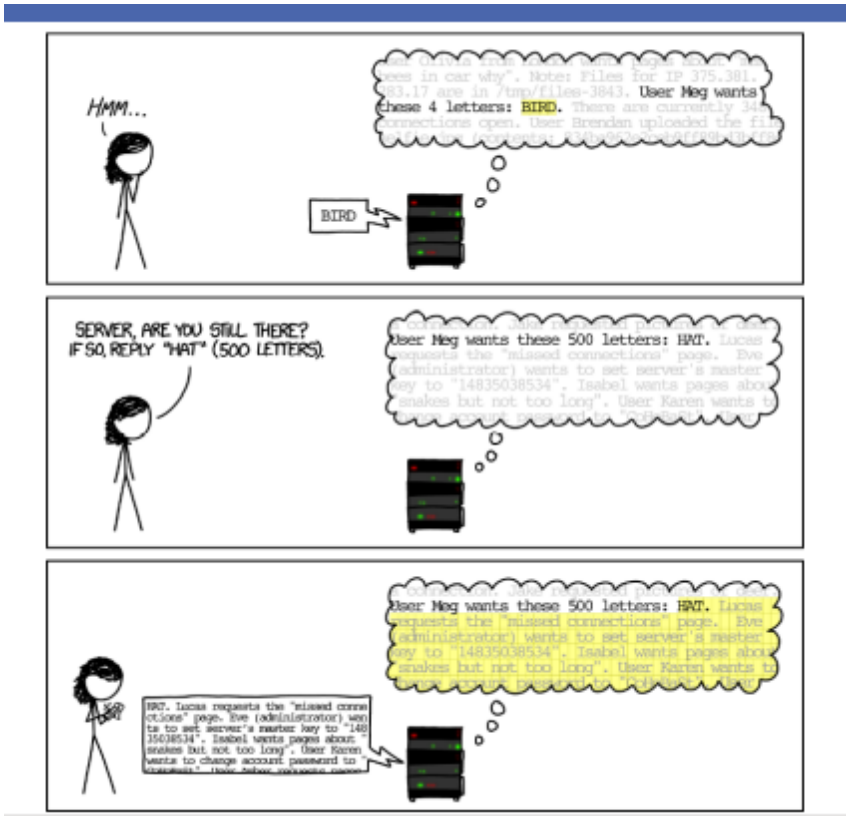
CORRUPTING DATA

Buffer overflow essentially affect code but overflowing data can be used as well to:

- modify a secret key to be one known by the attacker
- modify state variables to bypass authorization checks
- modify interpreted strings used as part of commands

For example, the heart bleed bug was a read overflow bug in openssl where the user can send to the server a request to understand if the server is active. The server saves this message and then sends it back to the user according to the length sent by the user.
The idea is that the user message specifies the length of the message it sends and the server returns the message obtained according to this length.





Stale memory is based on the fact that when a dangling pointer bug occurs when a pointer is freed, the program continues to use it.
So basically the attacker can use this freed memory to reallocate with values under his control.

Format string vulnerabilities.

Format function: C's printf family (printf, fprintf, sprint, etc.) supports formatted I/O

Format string

```
void print_record(int age, char *name)
 {   printf("Name: %s\tAge: %d\n",name,age);
 }
```

- Position in string indicates stack argument to print!

Format string parameter: Specifier that indicates type of the argument

- %s = string
- %d = integer
- etc.

Now the idea is that if we pass as a string a sequence of others %s then the program will try to get these values from the stack.

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Attacker controls the format string

- for each %S we use a value is retrieved from stack and handled as an address
- if the address belongs to the program space then it is printed
- if not an exception is generated.