Sicurezza informatica

Federico Zhou

December 3, 2021

Data	Argomento	Materiale didattico	Tipo attivita	Altro
11/03/2021	Low-Level attack: Buffer Overflow (Stack-Based) Part 1	Slide Buffer Overflow Aleph One article on Buffer Overflow Shellcoder Handbook	Didattica frontale	
18/03/2021	Low-Level attack: Buffer Overflow Practical Part 2	GDB Cheat Sheet Peda Repository	Didattica frontale	
25/03/2021	Low-Level attack: Heap Overflow on Metadata	Heap Overflow on Metadata Heap Overflow Repository on Metadata	Didattica frontale	
08/04/2021	Low-Level attack: Use After Free (UAF)	User After Free Vulnerabilities	Didattica frontale	
15/04/2021	Low-level defense: Memory safety and Type safety	Defense against low-level attacks Low-fat pointer technique LowFat Pointer Implementation GDB Input handle	Didattica frontale	
22/04/2021	Low-level defense: Canary, ASLR, DEP, Return Oriented Programming (ROP)	Defense against low-level attacks Return Oriented Programming (ROP)	Didattica frontale	
29/04/2021	Low-level defense: Return Oriented Programming (ROP)	ROP Gadgets Tool Blind ROP paper	Didattica frontale	
06/05/2021	Low-Level defense: Control Flow Integrity (CFI)	Control Flow Integrity (CFI) Control Flow Integrity (CFI) paper	Didattica frontale	
13/05/2021	Program Analysis Testing for Security Purpose	Security Analysis Introduction	Didattica frontale	
13/05/2021	Program Analysis I: Symbolic Execution (Introduction)	Symbolic Execution	Didattica frontale	
20/05/2021	Program Analysis II: Symbolic Execution	Symbolic Execution	Didattica frontale	
20/05/2021	Program Analysis II: Z3 Theorem Prover	Z3 (Microsoft) Theorem Prover Z3 github Documentation Z3 python example Z3 example	Didattica frontale	
27/05/2021	Program Analysis II: Fuzzing	Fuzzing Techniques Klee Tutorial 1 Klee Tutorial video Qsym Hybrid Fuzzer	Didattica frontale	
03/06/2021	Side Channel Attack	Meltdown & Spectre	Didattica frontale	

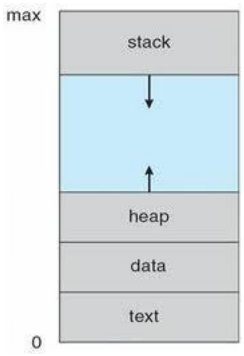
1 Buffer overflow

A buffer overflow is a bug that affects low-level code, normally a program would crash but an attacker can alter the situation *to steal informations, corrupt valuable information or run an attacker's code of choice.*
They are relevant because C and C++ are still popular, and buffer overflow attacks stil happen regularly. They have a long history and there are many approaches to defend against them.
There levels in which we can act to defend against them, the compiler, the OS, or the architecture.

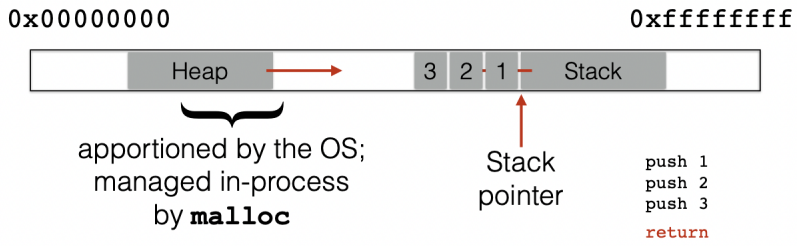
Memory Layout (32bit, Linux)

In pile in a 32 bit system, Linux or Intel 32-bit the addresses range from 0x00000000 to 0xBFFFFFFF.

The stack, which grows downwards, is the place in which activation records are stored, together with the heap they are created in runtime. The heap is managed with malloc, calloc, free and grows upwards, and contains variables. The text segment contains the instructions.



Memory operations make the memory grow upwards, this makes overwriting the SFP possible, allowing memory errors.



Stack and functions: Summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump to the function's address**

Called function:

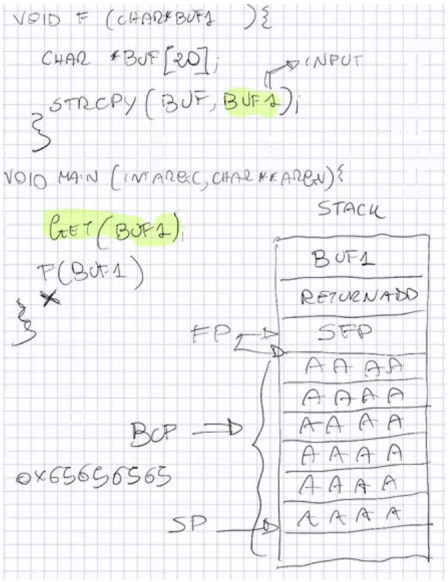
4. **Push the old frame pointer** onto the stack (%ebp)
5. **Set frame pointer** (%ebp) to where the end of the stack is right now (%esp)
6. **Push local variables** onto the stack

Returning function:

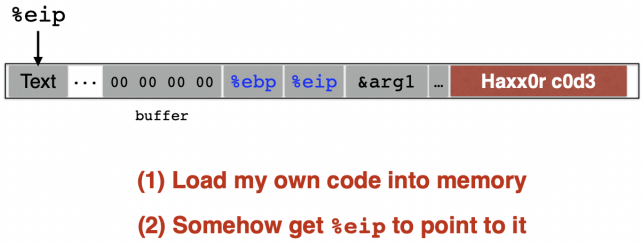
7. **Reset the previous stack frame**: %esp = %ebp, %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%esp)

Firstly let's talk about buffers, buffers are a contiguous piece of memory associated with a variable or a field, these are common in C. By overflowing a buffer we can put more into

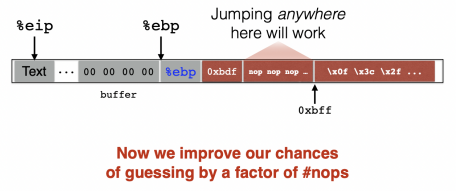
the buffer than it can hold.
In this example: BUF1 (which is big [24], is copied into BUF[20], since memory operation increases the heap upwards, it is possible to overwrite the SFP, causing a segmentation fault error.

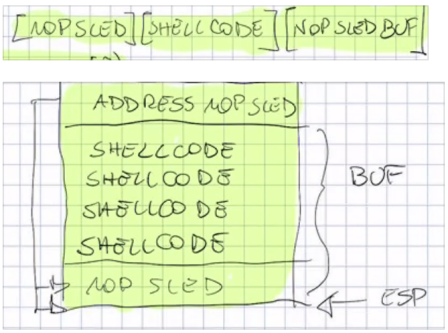


We have crashed the system, but how can we exploit this?
Note that strcpy will let you write as much as you want till a terminator character, we could introduce particularly crafted code to wreak havoc.
If instead of AAAA (which is: 0x65656565) we introduce code which alters the normal execution of the program, overwriting the return address.
How do we build the injection vector?



Note that the "disallocation" only consists in a subtraction of the reference pointer, the data remains
We introduce arbitrary shellcode, and by hijacking the return address we can execute code we want
We can only load machine code into memory, which are instructions already compiled and ready to run, since we're trying to give the attacker general access to the system, the best candidate is the general-purpose shell.





The shellcode is carefully crafted in the format *[SHELLCODE][ADDRESSBUF] = Attack Vector* and it is copied with the function call "strcpy". It is fundamental we get to know the [ADDRESSBUF], since if we don't an unrecognized instruction is going to run and C is gonna throw an IllegaArgumentException.

Shellcode: machine code -that we want to execute- to run a shell

The idea: we can't insert a "jump to my code" instruction, so we hijack the saved %eip. Since we don't know the desired %eip address we'll have to:

- Try a lot of different values, upwards of 2^{32} possible answers

- Without address randomization, the stack always start from a fixed address, and it doesn't grow very deeply.

How do we increase the chances to execute our shellcodes ?
Introduce the NOPSLED: 0x90 is a NOP instruction, single byte instruction which does nothing. We can fill our vector buffer of NOP operations, thus increasing the chances of execution Depending on the size of the buffer, we have to size our shellcode. The size depends on the allocated space by the target char array in strcpy.
char BUF[20]; strcpy(BUF, BUF1); size of the buffer to work with is 20
The computer by itself has protections, mainly towards the **returnAddress**, such as canary, nx

Heap vs Stack

Let's take an overlook over the differences between the Stack and the Heap:

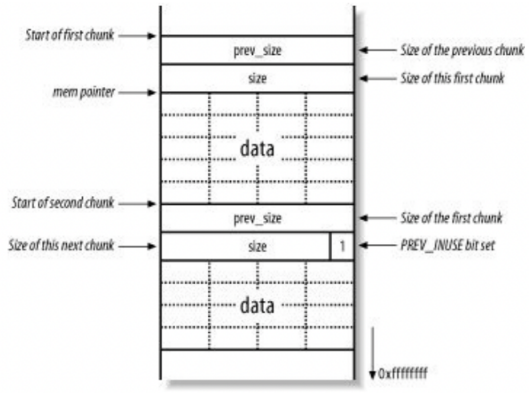
- The stack:
  - has a fixed memory allocation, which is only known at compile time.
  - inside it reside local variables, return addresses, functions args.
  - it's very fast and automatic, and it is managed by the compiler.

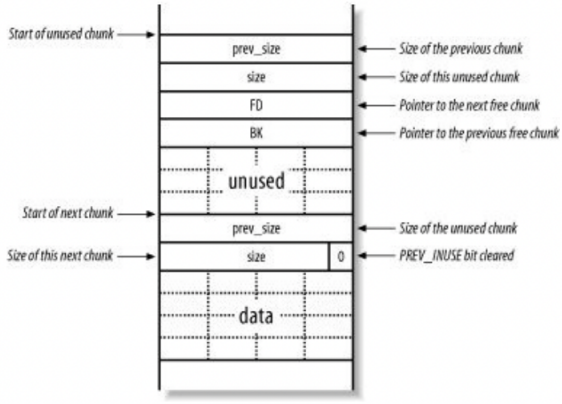
- The Heap:
  - has dynamic memory allocation, known at runtime.
  - inside it reside objects, big buffers, structs, persistence and larger objects
  - it's slower, manual, managed by the programme

The Heap is a pool of memory used to dynamic allocations, managed with:
- *malloc()*: to allocate memory on the heap
- *free()*: to release memory on the heap

```
struct malloc_chunk {
 INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
 INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

 struct malloc_chunk* fd;         /* double links -- used only if free. */
 struct malloc_chunk* bk;

 /* Only used for large blocks: pointer to next larger size.  */
 struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
 struct malloc_chunk* bk_nextsize;
};
```

Let's see an overview of a heap chunk:





The chunk is a part of the memory allocated in the heap. Alongside the data, it also contains metadatas:

- prev_size: contains the size of the pervious chunk

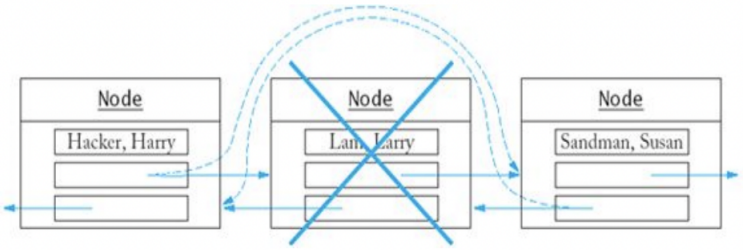
- size: it is also equipped with a bit-switch *PREV_INUSE*. It contains the size of the previous chunk.
- FD: the pointer to next free chunk
- BK: the pointer previous free chunk

The heap grows upwards. *prev_size and size* concurrently form the metadata section of the heap chunk. When we call $P = malloc(128)$, the pointer points to memPointer, just below the metadatas.

The free function When *free(p)* is used, the LSB of the size field in the metadata of the next chunk is cleared. Additionally the prev_size field of the next size will be set to the size of the chunk we are currently freeing.
The system keeps multiple lists of the freed chunks, which can be of different size, when a chunk is freed, we can verified if the previous chunk is freed to, and if it is we can unify them into a single bigger chunk.
When an allocation request is made, the system looks into the first free chunk that has a size $\geq$ than the size requested. If no free chunk is found, the top chunk will be used.

AV_TOP è il riferimento al top chunk; rappresenta lo spazio rimanente. Quando solleviamo una memory request e lo spazio nello heap è esaurito *eg. malloc(256)* il topchunk è diviso in due, e rispettivamente le parti diventano topchunk e la parte richiesta.
When we deallocate a chunk, we use an unlink function, this leaves an opening for us, the attacker. By hijacking the two metadata values it allows us to write an arbitrary value to an arbitrary location.

```
void unlink(malloc_chunk *P, malloc_chunk *BK,
malloc_chunk *FD)
{
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```



the intended process

```
...
FD->bk = return address;
FD = P->fd;
BK = P->bk = address of the buffer (injection vector);
FD->bk = BK;
...
```

malicious code which allows us to manipulate the BK

Unfortunately this technique no longer works, the current unlink function verifies beforehand the values We can however look for other flaws:

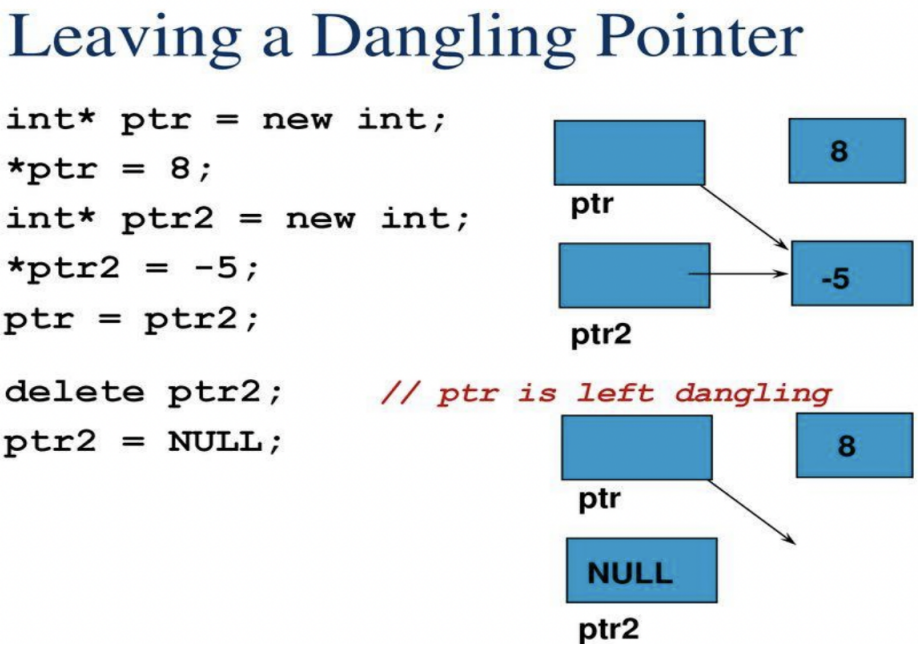
House of Force vulnerabity

Conditions:

- It requires 3 malloc, more precisely a malloc that allows us to overwrite the top chunk, one malloc with a user controllable size and another call to malloc.
- 1 (or 2) strcpy

Use after free vulnerability

Use after free vulnerability are vulnerabilities that happen when a pointer to an object that hash been freed is deferenced. This can lead to information leakage, and furthermore the attacker could also modify unintended memory locations that can potentially lead to code execution.
A prime candidate for an attack is a dangling pointer. Dangling pointer happen when a pointer, which was previously the target of another pointer is deleted, leaving the other pointer dangling.



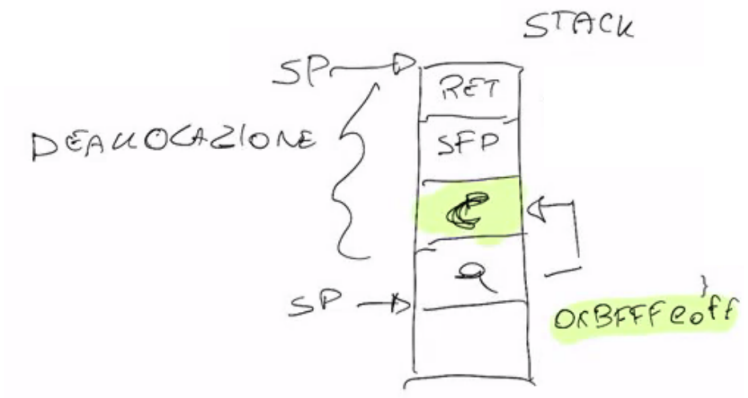
This happens because in C the entire management of pointers is left to programmers. In java this issue doesn't manifest itself because unused/unallocated memory spaces are collected by the garbage collector.

Let's see an actual example of user after free vulnerability

```
char *retptr() {

char p ,*q ;
q = &p ;

return q ; /* deallocation on the stack */

}
```

```
int main( ){

    char *a , *b;
    int i ;

    a = malloc(16) ;
    b = a + 5;

    free(a) ;
    b[2] = 'c' ; /* use after free */

    b = retptr( ) ;
    *b = 'c' ;    /* use after free */

}
```

In this example, a is a contiguous memory space long 16 bytes, b points to a location inside of a.
b, after using *b[2] = 'c'* is able to modify a memory portion of a after it has been freed (de-allocation).



b punta a una porzione dello stack che è stata deallocata. chiamando *b = 'c' modifichiamo una porzione dello stack che è stata deallocata precedentemente (nello stack p diventa c).

Defence and mitigation mechanisms

All these attacks have in common a couple of characteristics:

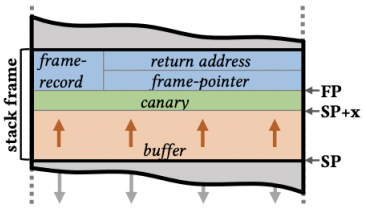
- The attacker is able to control some data that is used by the program
- The use of data permits unintentional access to some memory area in the program past a buffer, or to an arbitrary position on the stack.

So what can we do to *mitigate* these vulnerabilities?
First of all it's important we mention that we are not going to implement solutions which are going to leave us with a type safety and memory safety language. We are only going to implement mechanisms which are going to make the exploitation of these vulnerabilities too costly/time consuming.

Type safety and memory safety are two fundamental mechanisms which can help us mitigate these vulnerabilities, these properties, if met, ensure an application is immune to memory attacks. Languages such as C and C++ don't implement memory and type safety due to its efficiency load, other languages that do are JAVA or Python.

So what can we actually do?

- Automatic defences:
  these defences are specific to the compiler and help protect the return address and -
  Stack canaries: protects the return address from hijacking.
  More precisely a stack canary is a value placed on the stack such that a stack-buffer overflow will overwrite it before corrupting the return address. The buffer overflow can then be detected by verifying the integrity of the canary before performing the return. It is fundamental that the stack canary remains confidential, otherwise the attacker could just copy it and hijack the control flow anyway



- Address space layout randomisation **(ASLR)**: is a memory-protection process for operating systems that guards against buffer-overflow attacks by randomising the location where system executables are loaded into memory.

Being able to determine beforehand the position of processes and functions in memory makes exploitation easier, and ASLR is able to put address space targets in unpredictable locations, the attacker won't be able to identify the correct address space, making the application crash and notifying the system.

- Return-oriented programming **(ROP)**attacks are based on the technique that allows the attacker to hijack the program control flow by executing machine code instructions present in the machine's memory.
  This is done by manipulating the call stack by exploiting vulnerabilities in a program, typically a buffer overflow.
  Control Flow Integrity **(CFI)** are mechanisms implemented to prevent a variety of attacks based on the redirection of the control flow of the program.

- Secure coding are a set of practises that applies to security considerations that help defend against vulnerabilities, bugs and exploits. Secure coding introduces safeguards which reduce or eliminate the risk of leaving security vulnerabilities in the code, most important of which are **memory safety and type safety**

Memory Safety

Memory safety is a fundamental principle which helps mitigate C's vulnerabilities, in particular to guarantee a degree of memory safety in C we can impose to:

- only create pointer through standard means

  p = malloc(...), or p = &x, or p = &buf[5] etc.

  item only use pointers to access memory that "belong", that is within the bound of that pointer and of the type that corresponds. (int* to integers, char* to characters).

This essentially combines two principles: *temporal safety and spatial safety.*

Spatial Safety (in the heap)

Pointers are viewed as triples (p, b, e) where:

- p is the actual pointer
- b is the base of the memory region it may access
- $e$is the extend (or the bounds) of the region

The deferenced access shall be allowed iff $b \leq p \leq e - sizeof(typeof(p))$.
An implementation of a spatial safety compliant memcpy:

```
1  void memcpy(void *dst, void *src, int n)
2  {
3    void *dst_base = base(dst);
4    size_t dst_size = size(dst);
5    void *src_base = base(src);
6    size_t src_size = size(src);
7    for (int i = 0; i < n; i++) {
8      void *dst_tmp = dst + i;
9      void *src_tmp = src + i;
10     if (isOOB(dst_tmp, dst_base, dst_size))
11         error();
12     if (isOOB(src_tmp, src_base, src_size))
13         error();
14     *dst_tmp = *src_tmp;
15   }
16 }
```

Fig. 2. Instrumented version of simple `memcpy`.

Temporal safety

Memory regions can either be defined or undefined:

- defined means allocated (and active)
- undefined means allocated, uninitialised, or deallocated
  A temporal safety violation occurs when an attacker tries to access undefined memory space
- spatial safety assures that an accessed region is legal
- temporal safety assures that region is still in play

When a free function is called, the pointer towards that region is freed up, but the data remains in the stack, just unreferenced. The garbage collection's job is the memory management, its main function is to reclaim memory which was allocated by the program, but is no longer referenced.
The aim is to implement mechanisms similar to the garbage collector, but not quite; we want to protect vulnerable portions of the memory. The garbage collectors are demanding in terms of resources.

But why would we want to use a language that is **not memory safe and type safe**?
The easiest way to avoid all these vulnerabilities would be to use a **memory safe language**, languages that are **type safe are even better.**

Type safety > Memory Safety

Type safety in C

The main reason is that C and C++ are **here to stay**, while not memory safe, writing memory safe programs is possible.

The problem of type safety and memory safety has been known for more than 20 years, the limiting factor has always been **performance** (around 12x and 17x overhead). Furthermore adding type safety would also make C and C++ much slower.
Type safety enforcement is expensive, the most important mechanisms are

- **Garbage collection** which avoids temporal violations, but uses much more memory
- **Bound** and null pointer checks which avoids spatial violations
- **Hiding representation** may **inhibit optimisation**
  technique such as C-style casts, pointer arithmetic, & operators would not be allowed

So what can we do in C?

- Compiler could add code to **check for violations**
  An out-of-bounds access would result in immediate failure just like an *ArrayBoundsException* in Java.
- **No dangling pointers**
  Accessing a freed pointer violates temporal safety, accessing uninitialised pointers is also not OK:

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

```
int *p;
*p = 5; // violation
```

- **Type safety**
  Each objects shall be ascribed a type (int, pointer to int, pointer to function). Operations on he objects shall be always compatible with the object's type. Type safe programs can't encounter runtime errors. Type safety implies that objects shall always have a compatible types, while guarantee bounds compliance

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe, but not type safe

So how can we guarantee a degree of type safety in C?

Using **enforce invariants**, are properties which cannot be violated by type, by enforcing invariants in the program. Most notably is the enforcement of **abstract types**, which characterised modules. It keep the **representation of the modules hidden from others**. This guarantee a **degree of isolation** from the rest of the system.

Type-enforced invariants can relate to security properties, by expressing stronger invariants about data's privacy and integrity we can mitigate type mismatch exploits, such as the following in Java.

- **Example**: **Java with Information Flow (JIF)**

```
int{Alice→Bob} x;
int{Alice→Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is not
       //as strong as x
```

Types have *security labels*

Labels define what information flows allowed

All of these mechanisms concurrently compose the security by design principle, which means that the language and its capabilities are foundationally secure. In this approach security is built into the system, not mitigating since the design stage notable vulnerabilities. There are already languages which provide similar features to C, while remaining type safe, notable examples are Google's Go, Mozilla's Rust, Apple's Swift.

Other defensive strategies:

These are complementary, they mitigate bugs but can't shut them out entirely

- Make the bug harder to exploit: introduce more steps, to make them difficult or impossible
- Avoid the bug entirely: practice secure coding practices, advanced code review and testing

To implement these, let's recall the steps of a stack smashing attack:

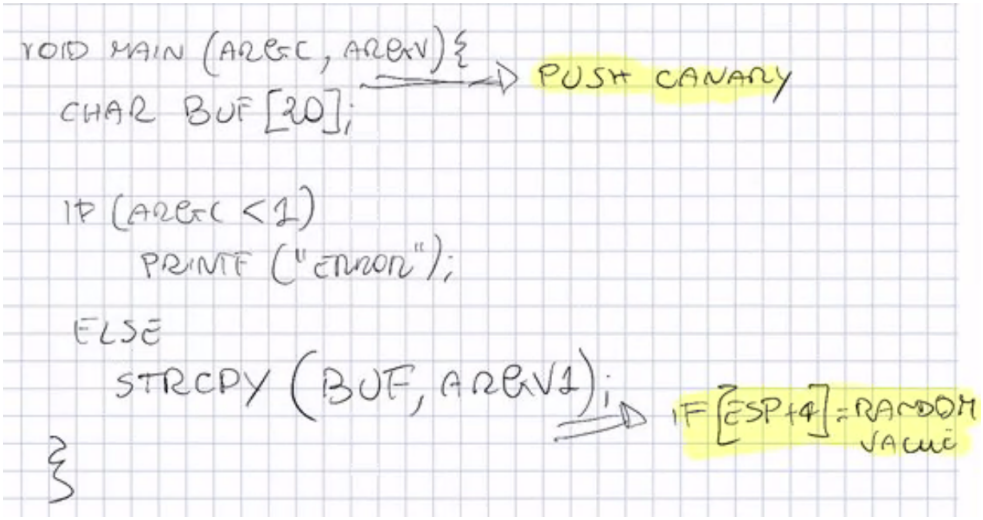
- Putting the attacker code into the memory. The shellcode must also not contain zero (it is interpreted by the machine as a string terminator)
- Getting *%eip* to point at the shellcode
- Finding the return address (by guessing the raw address)

To make these attacks steps more difficult we can use libraries, compiler mechanisms and the operating system. We are trying to fix the architectural design, and not the code.

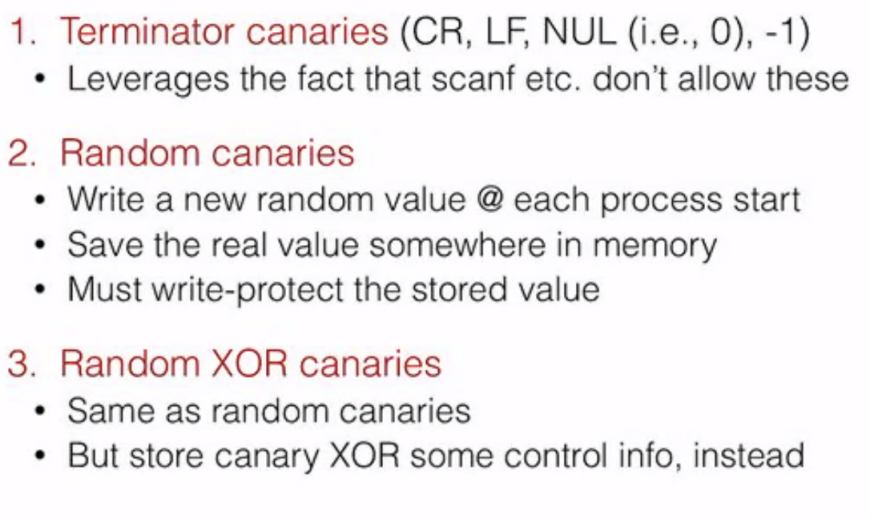
Overflows detection

Stack canaries

Managed by the compiler, canaries work by protecting the return address by detecting any change that could have harmed the stack's integrity. The canaries are placed inside the stack in a protected memory section between user and system variables.
The canary's value has to be random, and unaccessible by the attacker otherwise it could be overwritten with the same value. The canary is introduced when the function is called, using a machine code function called *push canary*



Before calling the *ret* the value of the canary is checked, raising errors if it's not correct. The values the canary can assume are these:



3. Random XOR canaries work by doing:

    1. $[ret\ XOR\ canaryValue]$ to produce che canary value

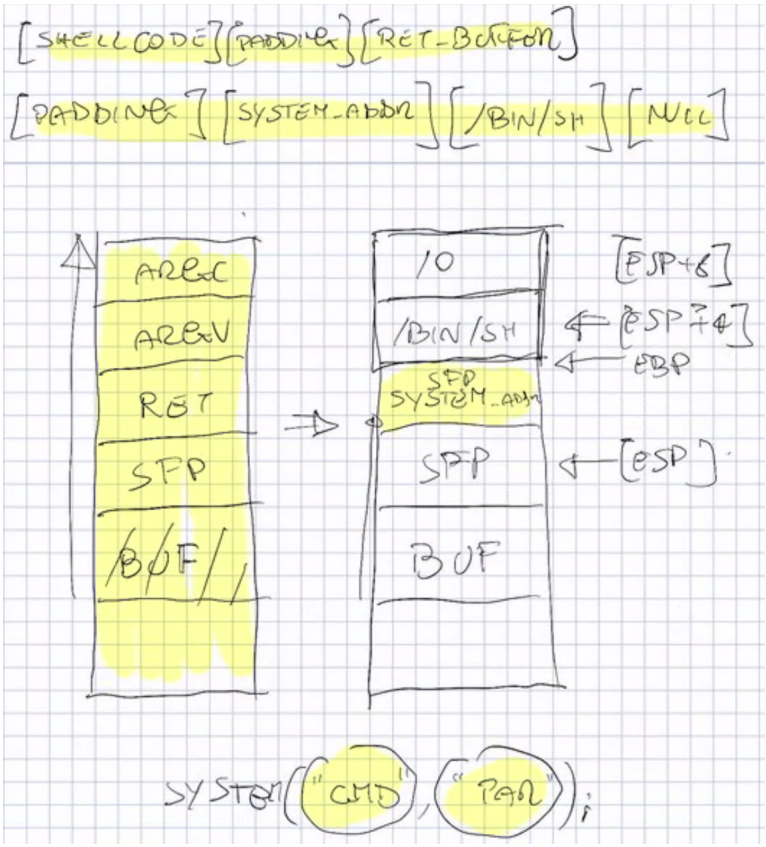
    2. $[ret\ XOR\ canaryValue]$ the verify the canary value

This works due to the nature of XOR, calling it two times gives us the initial value

DEP — Data Execution Prevention

This defence works by making the stack and heap memory space non-executable. Attackers bypassed this defence by having the shellcode point to system libraries in an attack called *return-to-libc*.
lib-c libraries are used by the gcc compiler, meaning they are always available.

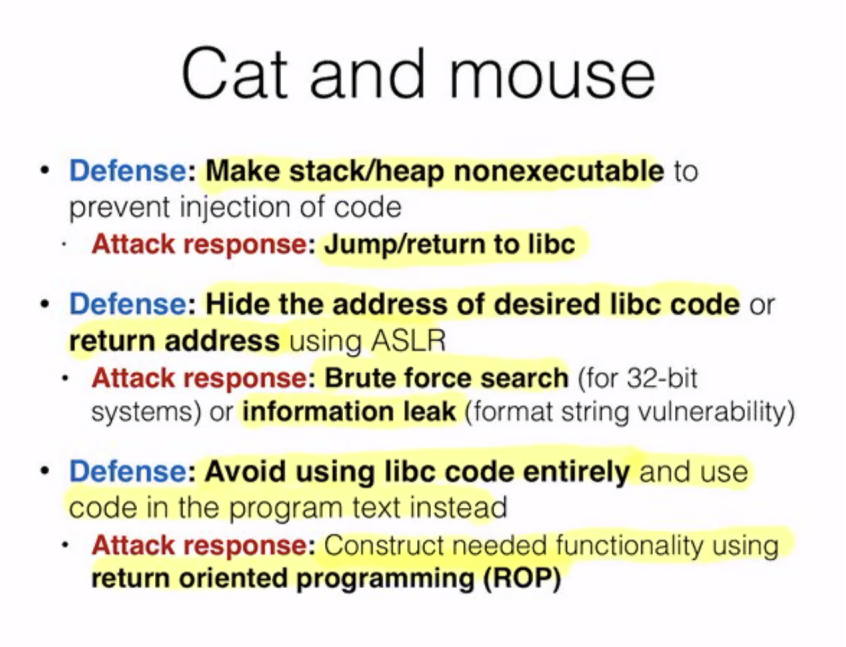
To use this attack, system is called. System is a function that has argument a terminal command. In order to work with this vulnerability, we need to first: call the system function, and two: have it run a meaningful command. This is done by placing the command next to the system call. The injection vector is modified in the following way.



Defence strategies mitigated the *return-to-libc attack* by introducing *Address-space Layout Randomitazion or ASLR* which places standard libraries and other elements in memory in random places, making them harder to guess. It also has some caveats:

- Only shifts the offset of memory areas, not particular locations within those areas
- May not apply to program code, but only libraries
- Need sufficient randomness, otherwise it could be brute forced.
  this makes this technique more promising on 64-bit systems

When we talk about *cat and mouse* we mean the game of cat and mouse being played by attackers and the defence specialists.

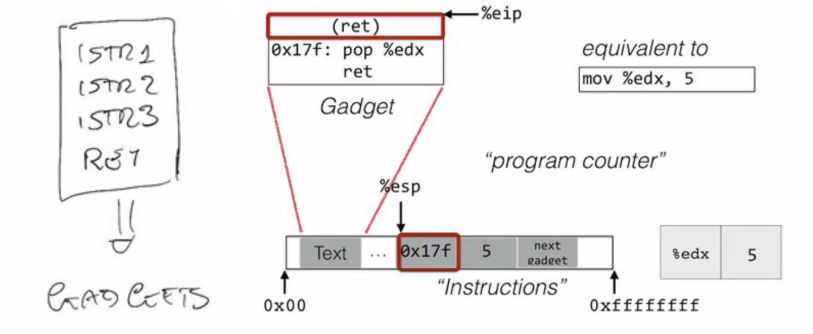


Return oriented Programming

First introduced by Hovav Shacham in 2007, the idea is: rather than use a single libc function to run the shellcode, we string together pieces of existing code called gadgets to do it. Intuitively, we need to find the gadgets we need, and we need a mechanism to string them together

Gadgets

Gadgets are instruction groups that end with *ret*:



The memory structure that we're using is still the stack, which is also where we're loading our code.

- %esp serves as program counter
- *gadgets* are invoked via *ret* instruction
- *multiple gadgets* are linked with each others using *ret* instructions
- *gadgets* get their arguments via *pop, peek etc.*

In this example we're trying to find an applicable instruction equivalent to *mov %edx, 5*, being: *pop %edx ret*.

So how do I actually find gadgets?

One approach would be an automated search of the target binary for gadgets. This technique works by looking for *ret* instructions, by working backwards.

Are these gadgets actually enough to do anything interesting?

Yes, Shacham found that with significant codebases (such as libc) gadgets are *Turing complete* meaning they're theoretically capable of solving any computational problem. Scwartz, in 2011, automated the gadget shellcode creation process (ROP compiler), without needing Turing completeness.

In CISC there is a degree of variability in the length of instruction (which is not present in ARM for example since it's a RISC)(RISC instruction lengths are fixed) which provides an extensive degree of freedom to exploits gadgets. This particular aspect is called dense Instruction Set.

```
1   int main(int argc, char **argv)
2   {
3     int num;
4
5     scanf("%d", &num);
6
7     if(num == 1234)
8       printf("ok\n");
9     else
10      printf("no\n");
11
12    return 0;
13  }
```

```
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
call   8048304
call   8048360
call   8048490
leave
ret
pushl  0x80495dc
jmp    *0x80495e0
add    %al,(%eax)
jmp    *0x80495e4
push   $0x0
jmp    8048290
jmp    *0x80495e8
push   $0x8
jmp    8048290
jmp    *0x80495ec
push   $0x10
jmp    8048290
jmp    *0x80495f0
push   $0x18
jmp    8048290
xor    %ebp,%ebp
pop    %esi
mov    %esp,%ecx
and    $-16,%esp
push   %eax
push   %esp
push   %edx
push   $0x80483e0
push   $0x8048430
push   %ecx
push   %esi
push   $0x8048384
call   80482c0
hlt
nop
nop
push   %ebp
mov    %esp,%ebp
```

```
push   %ebx
sub    $0x4,%esp
call   8048310
pop    %ebx
add    $0x12c8,%ebx
mov    -4(%ebx),%edx
test   %edx,%edx
je     8048326
call   80482d0
pop    %eax
pop    %ebx
leave
ret
nop
...
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
cmpb   $0x0,0x8049600
je     804834b
jmp    804835d
add    $0x4,%eax
mov    %eax,0x80495fc
call   *%edx
mov    0x80495fc,%eax
mov    (%eax),%edx
test   %edx,%edx
jne    8048341
movb   $0x1,0x8049600
leave
ret
nop
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
mov    0x8049508,%eax
test   %eax,%eax
je     8048381
mov    $0x0,%eax
test   %eax,%eax
je     8048381
movl   $0x8049508,(%esp)
call   *%eax
```

```
leave
ret
nop
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $-16,%esp
mov    $0x0,%eax
add    $0xf,%eax
add    $0xf,%eax
shr    $0x4,%eax
shl    $0x4,%eax
sub    %eax,%esp
lea    -4(%ebp),%eax
mov    %eax,0x4(%esp)
movl   $0x80484e8,(%esp)
call   80482b0
mov    -4(%ebp),%eax
cmp    $0x4d2,%eax
jne    80483cb
movl   $0x80484eb,(%esp)
call   80482a0
jmp    80483d7
movl   $0x80484ee,(%esp)
call   80482a0
mov    $0x0,%eax
leave
ret
nop
...
push   %ebp
mov    %esp,%ebp
push   %edi
push   %esi
push   %ebx
call   8048483
add    $0x11ed,%ebx
sub    $0xc,%esp
lea    -224(%ebx),%eax
lea    -224(%ebx),%edi
sub    %edi,%eax
sar    $0x2,%eax
```

```
lea    -1(%eax),%esi
cmp    $-1,%esi
je     8048419
lea    0x0(%esi),%esi
call   *(%edi,%esi,4)
dec    %esi
cmp    $-1,%esi
jne    8048410
lea    0x0(%esi),%esi
call   80484c4
add    $0xc,%esp
pop    %ebx
pop    %esi
pop    %edi
pop    %ebp
ret
lea    0x0(%esi),%esi
push   %ebp
mov    %esp,%ebp
push   %edi
push   %esi
push   %ebx
call   8048483
add    $0x119d,%ebx
sub    $0xc,%esp
call   8048278
lea    -224(%ebx),%eax
lea    -224(%ebx),%edx
sub    %edx,%eax
sar    $0x2,%eax
mov    %eax,-16(%ebp)
je     804847b
xor    %edi,%edi
mov    %edx,%esi
lea    0x0(%esi),%esi
lea    0x0(%edi),%edi
inc    %edi
call   *(%esi)
add    $0x4,%esi
cmp    %edi,-16(%ebp)
jne    8048470
```

```
add    $0xc,%esp
pop    %ebx
pop    %esi
pop    %edi
pop    %ebp
ret
mov    (%esp),%ebx
ret
nop
...
push   %ebp
mov    %esp,%ebp
push   %ebx
sub    $0x4,%esp
mov    0x80494f8,%eax
cmp    $-1,%eax
je     80484bd
mov    $0x80494f8,%ebx
lea    0x0(%esi),%esi
lea    0x0(%edi),%edi
call   *%eax
mov    -4(%ebx),%eax
sub    $0x4,%ebx
cmp    $-1,%eax
jne    80484b0
pop    %eax
pop    %ebx
pop    %ebp
ret
nop
...
push   %ebp
mov    %esp,%ebp
push   %ebx
sub    $0x4,%esp
call   80484d0
pop    %ebx
add    $0x1108,%ebx
call   8048330
pop    %ecx
pop    %ebx
leave
ret
```

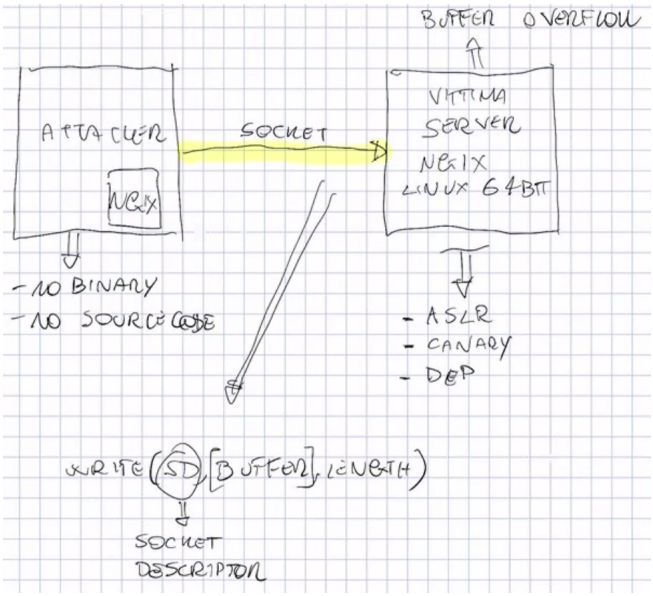
With a simple program like this, we can extract a gigantic amount of instruction.

Hacking Blind

Berkley students managed to attack victim server via a socket, which has all the defence mechanisms we have previously seen enabled, namely ASLR, canary and DEP. The attack happens based on a socket and a linux 64 bit server, via buffer overflow. The exploitation is based on the *write* function:

write(SD, [buffer], length)
SD is socket descriptor

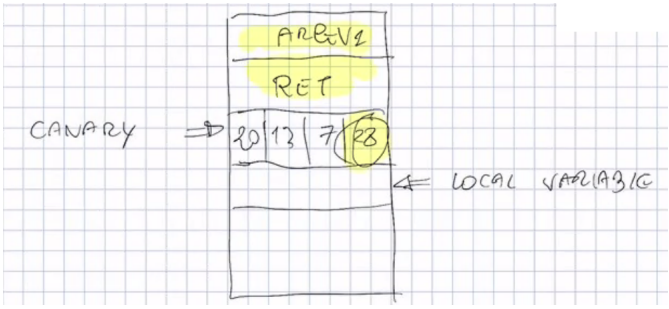
Utilising a flaw in the *write* function the attacker is able to copy the buffer portion of memory



A fully automated attack based on this exploit yields a shell in under 4000 requests (20 minutes) against a modern system.

Techniques needed

1. Generalised stack reading:
   the possibility to read the stack in its entirety, used to leak the canaries (replace them with the correct values), and to defeat the ASLR (if I am able to read the stack, I am also able to read the location of the program).
   The canary attack these students did is based on the fact that when a thread crashes on a server, only the thread is reset and not the entire system, and therefore the canary value remains the same. The attacker can progressively guess byte per byte the value of the canary. In this example the canary is of size 4 bytes, but on 64 bits system the canary is of size 8 bytes.



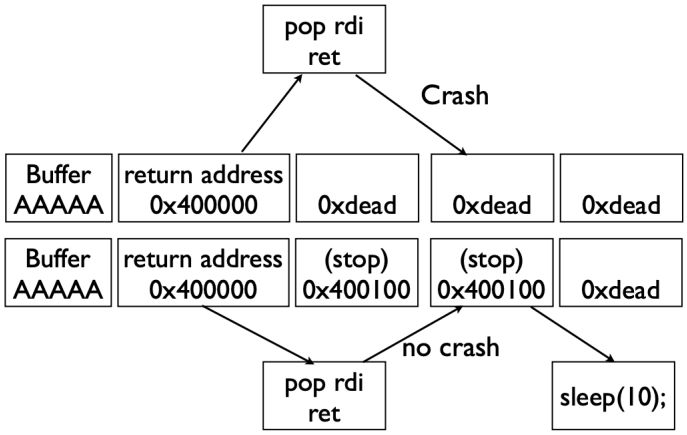
To guess the canary we try every value from *0 to 256* every byte, when the thread crashes we proceed with the next value, if the custom value doesn't invoke a crash, we found our value. Using this technique we can also guess the value of the return address.

2. Blind ROP: this technique remotely locates ROP gadgets (write functions)
   The goal is to find enough gadgets to invoke *write*. If we manage to do this, the binary can be dumped from memory to network finding even more gadgets in the process. As previously mentioned, the *write* system call takes 3 arguments: a socket, a buffer and a length. These are passed in *rdi, rsi, rdx* registers, and the system call is stored in the *rax* register. The following gadgets are therefore needed:

```
1) pop rdi; ret (socket)
2) pop rsi; ret (buffer)
3) pop rdx; ret (length)
4) pop rax; ret (write syscall number)
5) syscall
```

But how do we find these gadgets?

pop rdi, rsi, rdx are manageable, *pop rax and syscall* are the problematic functions, syscall is almost never called since this is usually done through libraries.
To effectively find gadgets, we modify progressively the return address, sorting them into crash gadgets, which cause the socket to crash, non-crash gadgets, which produce an arbitrary effect.



We further elaborate on non-crash gadgets, implementing probes, traps and stops. If the gadget produce the effect we want the stop is invoked and the gadget is selected.

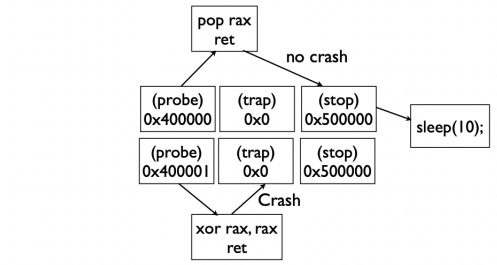


Figure 10. Scanning for pop gadgets. By changing the stack layout, one can fingerprint gadgets that pop words from the stack. For example, if a "trap gadget" is executed rather than popped, the program will crash.

To call functions the PLT (static loader table) and GOT (populated by the dynamic linker) tables are used at runtime to find libraries.

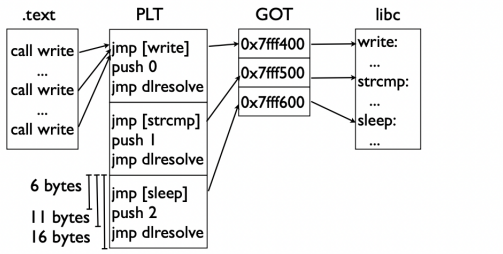


Figure 11. PLT structure and operation. All external calls in a binary go through the PLT. The PLT dereferences the GOT (populated by the dynamic linker) to find the final address to use in a library.

We have seen that memory safety and type safety solve most of the memory errors, *control flow integrity* is the last passive defence mechanism we see.

Control flow integrity

CFI aims to defend a program by observing the program's behaviour, if the program is not doing what we expect it to do, it might be compromised. To do this we need to define an **expected behaviour**, create mechanisms which let us **detect deviations from expectations effectively**, implement policies which **prevent compromise of the detector**

How efficient is CFI ?

Classic CFI imposes a 16% overhead on average, 45% in the worst cases, it works on arbitrary executables, and is not modular.
Modular CFI imposes a 5% overhead on average, and 12%in the worst cases, it works only in C (part of the LLVM, source code), and is modular, with separate compilation routines

How secure is CFI actually?

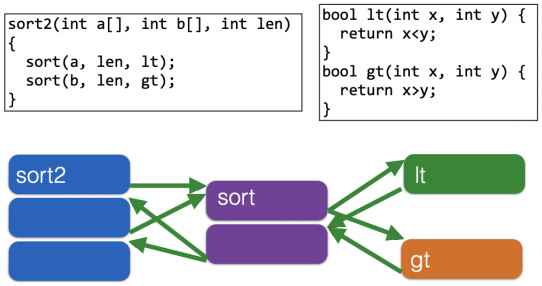
MFI can eliminate 95% of ROP gadgets on x86-64 versions of SPEC2006 benchmark suites. Average indirect-target reduction is of $> 99\%$, meaning that the percentage of possible targets of indirect jumps is almost completely ruled out.

Randomness and immutability of the detector

A degree of randomness and immutability is added to the detector to avoid compromise

Control flow graph — CFG

to define expected behaviour. One way we can do this is by utilising **call graphs**. The aim is to represent all the possible combinations of function calls to other functions



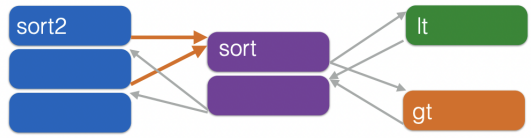
example of a function which sorts by calling 2 distinct functions

The code is broken up into **basic blocks**, **calls** are distinguished from **returns.** We compute call/returns CFG in advance, during the compilation or from the binary. We monitor the control flow of the program and ensure that it only follows paths allowed by the CFG.

Note that calls can be divided into direct calls and indirect calls: *direct calls jump to fixed addresses*, the target address cannot be changed; *indirect calls are calculated at runtime*, the value of the target address cannot be calculated beforehand, these are *jmp, call, ret.*
We only need to monitor **indirect calls**, direct calls need not be monitored since an attacker cannot change a fixed address.

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

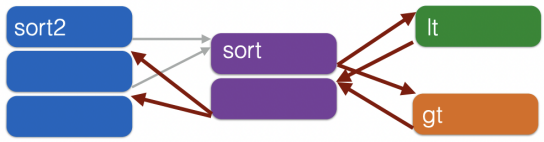
```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



direct calls always have the same target

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

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

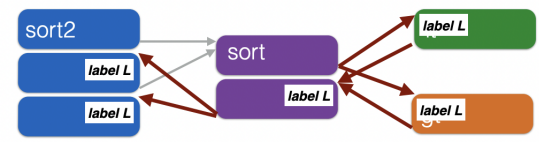


indirect transfers do not

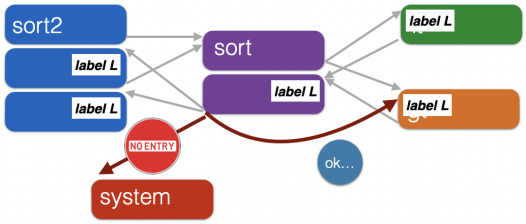
In-line reference monitor — IRM

IRM is used to detect deviations from expectations, it works by introducing control methods inside the program (in binary or inside the program) (program transformation). More particularly we utilise labels, determined by the CFG:

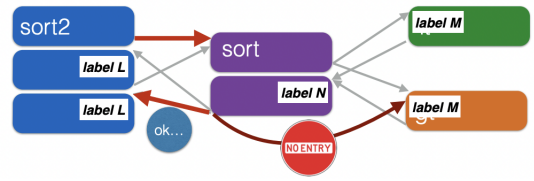
- insert a label just before the target address of an indirect transfer
- insert code to check the label of the target at each indirect transfer
  If the label does not match, we abort the execution



If for example an attacker manages to hijack the control flow of the program, when he tries to access a system call the jump is denied, since it's not validated by the CFG.



We can further improve simple labelling by introducing **detailed labelling**



- return sites from calls to *sort* must share a label L
- call targets *gt* and *lt* must share a label M
- remaining labels are unconstrained N

So how can we defeat CFI?

- inject code that has a *legal label*
  - won't work since we assume non-executable data
- modify code labels to allow a desired control flow
  - won't work since the code is immutable
- modify the stack during a check
  - won't work since the attacker cannot change register

CFI assurances

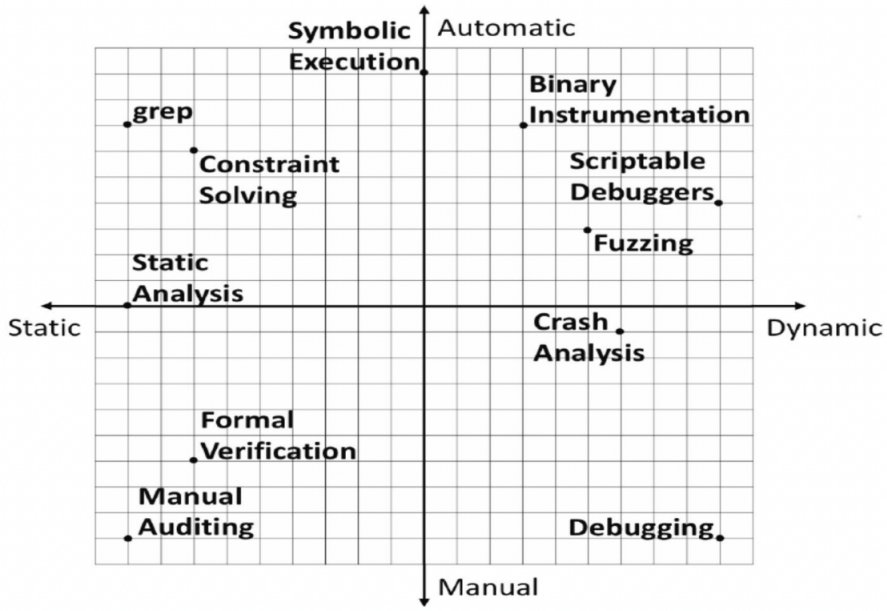
CFI defeats control flow-modifying attack such as remote code injections, ROP/return to libc, etc.

- It cannot defeat attacks based on manipulation of control flow that is allowed by the labels/graph, these are called **mimicry attack**. These attack mimic a valid control flow, bypassing detection.
- It cannot defeat data leaks or corruptions, heartbleed cannot be prevented, nor the *authenticated* overflow. In simpler terms, attacks based on the modification of attributes and variables are not preventable using CFI, such as:

```
void func(char *arg1)
{
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, str);
  if(authenticated) { …
}
```

Static Analysis and Dynamic Analysis — persa prima parte

Analysis can range from static to dynamic, and from automatic to manual. The main advantage of static analysis is that when designed, it's fast to execute; the downside is that the precision is not that great, potentially leading to false positives. Dynamic analysis is more precise than static analysis, and is done by actually executing the program and closely monitoring the execution. The downside of dynamic analysis is that only a single input can be verified at a time, making it slow (as compared to static)



main vulnerability detection techniques

The most common technique is debugging, monitoring the system execution looking for bugs and flaws. Symbolic execution is an example of testing that can be done both statically, and dynamically. Crash analysis aims to crash the program intentionally, and analyse the system's dump (root cause analysis).

Detection of the vulnerability

The aim of this process is to detect the presence of vulnerabilities in the code during the stages of development, testing, and maintenance.

Soundness and completess

While considering the tools available to us we need to consider the trade-off between *soundness* and *completeness*.

A detection technique is considered **sound** if for a given category, it can correctly conclude that a given program or system has or hasn't vulnerabilities (per una data categoria è in grado di dirmi se il programma è sicuro — eg. niente bof).
An unsound detection technique, on the other hand, may have **false negatives**, meaning that actual vulnerabilities may slip through the tool's detection techniques.
Achieving soundness requires a consideration based on all execution of a program. This can be done by static checking of the program code, and building up an abstraction of the program execution. Using this abstraction we aim to execute the program in all its possible permutations.

A detection technique is **complete** for a given category if any vulnerability if finds is an actual vulnerability. By contrast and incomplete detection may have **false positives**, it may therefore detect issues which aren't actually vulnerabilities.
Achieving completeness can be done by executing the program that show the vulnerability, this analysis technique should provide concrete inputs that triggers a vulnerability. Software testing is a way a developer can achieve completeness, by writing test cases with concrete inputs, and specific checks for the outputs

In the real world techniques are mixed and used concurrently to analyse the program, employing both static and dynamic analysis to achieve a good trade-off between soundness and completeness. An hypothetical tool may be able to achieve both soundness and completeness, unfortunately such a tool doesn't exist, so we try to balance in the best way we can.

Static analysis and Symbolic execution

Static analysis is any offline computation that inspects the code and produces results based on the quality of the code. This can be applied both for source code and binary code. Techniques include regular expressions, classic compiler flow analyses or some combination of these. It is important to note that static analysis does not have to use symbolic execution (input non concreti, ma associati a simboli).

Symbolic execution is a specific kind of off-line computation that computes some approximation of what the program actually does by constructing formulas representing the program state at various points. It is called symbolic because the approximation is usually based on some formula involving program variables and constraints on their values.

Symbolic execution

Testing is a technique widely used in software engineering and works by utilising assertions such as

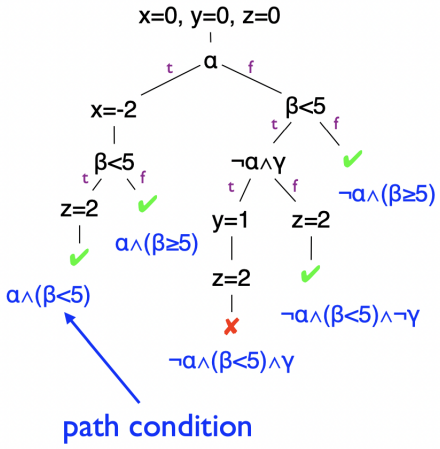
$$assert(f(3) == 5)$$

This technique is *complete*, meaning that a bug found is a real bug, but *not sound* since each test can only explore one possibility.
Symbolic execution generalises testing, the aim is to make employ a "more sound" technique that testing. Symbolic execution allows the implementation of unknown symbolic variables α in an evaluation such as

$$y = a; assert(f(y) == 2 * y - 1)$$

Symbolic execution example

```
1. int a = α, b = β, c = γ;
2.                  // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z != 3)
```



The path condition represents the input condition needed to reach a specific outcome. Each path condition can be used to create an actual input (usually done by the constraint solver). Soundness is partially achieved since we're able to verify for a number permutations of inputs. Each symbolic execution path stands for many actual program runs, as many concrete run under symbolic execution paths. We cover a lot more of the program's execution space than testing.
Static execution can therefore be viewed as static analysis which is complete, but not sounds; it is path, flow and context sensitive.

A bit of history

First implemented in the 70s, it didn't take off right away since symbolic execution can be quite computer intensive due to the high number of possible program paths, need to query

solve a lot to decide which paths are feasible and which assertions could be false. Computer were slower and smaller, not equipped with a lot of processing power. Nowadays, however, computer are faster and bigger, and symbolic execution is making a comeback also due to better algorithms and SMT/SAT (satisfiability modulo theories) solver, which can solve large instances very quickly

Symbolic variables

Symbolic variables are an extensions of the language to support expressions e to include symbolic variables, representing unknowns (but supported by the program itself)

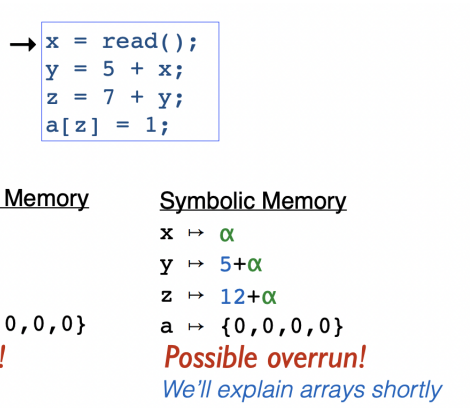
$$e ::= \alpha \mid n \mid X \mid e_0 + e_1 \mid e_0 \leq e_1 \mid e_0 \;\&\&\; e_1 \mid \ldots$$

- $n \in N$ = integers, $X \in Var$ = variables, $\alpha \in SymVar$

Symbolic variables are introduced when reading inputs, particularly when using *nmap, read, write, fgets etc.* If a bug is found, we can safely extract the input that produces the error. Furthermore we need to make or modify a language interpreter to be able to compute symbolically. normally a program's variables contain values, we modify them by also making sure they support *symbolic expressions.*

$$5, "hello" \rightarrow \alpha + 5, hello + \alpha etc.$$

Consideriamo il seguente programma:



the overrun comes from the fact that z has a value which exceed the array bounds of a

But in symbolic memory the value of α *can* exceed the bounds, but not necessarily

Path conditions

As previously said, path conditions are necessary conditions on inputs for reaching the target location is disjunction of all abstract path conditions. More specifically in programming a path condition diverges from another when an if is encountered, for example:

```
1  x = read();
2  if (x>5) {
3    y = 6;
4    if (x<10)
5      y = 5;
6  } else y = 0;
```

We represent the influence of symbolic values on the current path using a path condition π, particularly:

- line 3 reached when $\alpha > 5$
- line 5 reached when $\alpha > 5$ and $\alpha < 10$
- line 6 reached when $\alpha \geq 5$

Not all path are reachable, particularly we talk about *path feasibility* when we talk about whether a path is feasible

- line 3 reached when $\pi = \alpha > 5$, solution to which can be $\alpha = 6$
- line 5 reached when $\pi = \alpha > 5$ and $\alpha < 10$, but this is not satisfiable, the value of α cannot be both bigger than 5 and smaller than 3
- line 6 reached when $\pi = \alpha \geq 5$, solution to which can be $\alpha = 2$

Paths and assertions

```
1 x = read();          π = true
2 y = 5 + x;           π = true
3 z = 7 + y;           π = true
4 if(z < 0)            π = true
5   abort();           π = 12+α<0
6 if(z >= 4);          π = ¬(12+α<0)
7   abort();           π = ¬(12+α<0) ∧ 12+α≥4
8 a[z] = 1;            π = ¬(12+α<0) ∧ ¬(12+α≥4)
```

Assertions like array bounds checks are conditionals, can therefore verifiable. A security analyst can for example check the value of the index of an array during testing. If there is a feasible path leading to the aborts, we can deduce that the program has a security flaw/bug, if there are no feasible aborts, the program is safe.

- to reach the line 5:

$$\pi = 7 + 5 + \alpha = 12 + \alpha \rightarrow \pi = 12 + \alpha < 0$$

- to reach the line 7 abort:

$$\text{negate the line 4 condition, add the line 6 condiiton:}$$
$$\pi = \neg(12 + \alpha < 0) \wedge (12 + \alpha \geq 4)$$

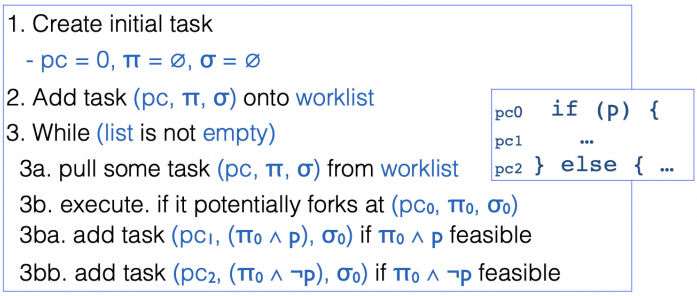
We look for path infeasibility to guarantee that there are no buffer overflow.

Forking execution

Symbolic executors can fork at branching points, which happen when there are solutions to both the path conditions and its negation. How can we systematically explore both directions?

- check feasability during execution and queue feasible path for later consideration
- concolic execution: run the program to completion, then generate new inputs by changing the path condition (esecuzione concreta del programma su dati input, con l'aggiunta dell'esecuzione simbolica)

To explore all possible paths we follow the execution algorithm:



this algorithm is based on a basic depth-search of a graph

Libraries and native code

At some point symbolic execution will reach the edges of a program such as libraries, systems or assembly code calls. An example can be the libc, which is insanely complicated, way too complicated for symbolic execution which can easily get stuck in it. We can introduce a simpler version of libc, for example newlib.

Concolic execution — Dynamic symbolic execution

Instead of running the symblic execution in a static way, we instrument the program to do symbolic execution as the program runs, initially the program considers concrete states (which could be randomly generated) to determine initial paths.
Each time a branch path is encountered, the expression is saved inside a *shadow path memory*, which are then explored one path at a time, from start to finish. The next path can be determined by negating some element of the last path conditions.
Concolic execution makes it really easy to concretise inputs, by replacing symbolic variables with concrete values that satisfy the path condition.

Search and SMT

Symbolic execution is appealingly simple and useful, but it's computationally expensive. Let's see how the effective use of symbolic execution boils down to a kind of search, and take a moment to see how its feasibility at all has been aided by the rise of SMT solvers. Basically the practical use of symbolic executiong boils down to

- searching for the paths
- solving the symbolic execution for these paths using SMT solvers while accounting for feasibility

One of the first thing to note is that we cannot run symbolic execution to exhausting, since this would quickly generate a exponential number of branches

```
1. int a = α, b = β, c = γ;    // symbolic
2. if (a) ... else ...;
3. if (b) ... else ...;
4. if (c) ... else ...;
```

3 variables, 8 program paths

And loops are even worse

```
1. int a = α;    // symbolic
2. while (a) do ...;
3. ...
```

potentially 2^{31} *paths*

Concolic executiong, as opposed to static analysis, doesn't terminate before considering all possible program runs. Static execution approximates multiple loop executions, assuming all branches or a number of loop iterations are feasible; this however can lead to false alarms. Concolic execution of the other hand aims to be more precise, granting more completeness.

Basic symbolic execution

So how do we actually look for the possible paths?
Simply, using seach algorithms, in particular

- Depth-first search (DFS) using a stack as worklist
- Breadth-first seach (BFS) using a queue as worklist

Potential drawbacks to this approach are that searches cannot be guided by any higher level knowledge (getting stucks in loops, path explosion etc.), moreover *DFS* can easily get stuck in parts of programs, making *BFS* the better choice between the two.

Advanced symbolic execution

As we have seen DFS and BFS are not optimal solutions, let's focus on prioritising searches, trying to steer searching towards paths more likely to contain assertions failures, and imposing a time limit on execution.
To this end, let's think of a program execution as a DAG in which

$$Nodes = programstates$$
$$Edges(n_1, n_2) = \text{transition from state } n_1 \text{ to } n_2$$

Intuitively we need a new graph exploration algorithm.

Randomness

Since we don't know a priori which path to take, we add a degree of randomness, we start by taking a concrete path, then executing a symbolic execution, randomly generating an expression n times, if no effects are detected we move on.

1. pick next path to explore uniformly at random
2. randomly restart search if we haven't hit anything interesting in a while
3. choose among equal priority paths at random

The problem with randomness is its non-reproducibility, we cannot reproduce specific execution unless we save the seeds.

Coverage guided heuristics

Coverage guided heuristics tries to keep a tab on previously visited paths using a score system. We visit statements we haven't seen before (or lowest score)

- we give a score to each statement based on the number of times it has been seen

- we pick the next statement to explore based on the lowest score

This technique tries to reach everywhere, even in hard to reach parts of the programs. The problem with this approach is that it may never be able to get to a statement if proper precondition not set up.

Generational search

Generational search is a hybrid of BFS and coverage-guided, this technique aims to increase the level of coverage

- Generation 0: pick one program at random, run to completion
- Generation 1: take paths from gen_0, negate one branch condition to yield a new prefix, find a solution for this prefix and take the resulting path
- Generation n: branch off again from gen_{n-1}

Generational search also uses coverage heuristic to pick priority

Combined search

Combined search aims to run multiple searches with different techniques at the same time alternating between them, it aims to increase parallelisation. The logic is since no-one-size-fits-alll solution, we switch between search techniques based on the occasion, selecting occasionally the algorithm.

SMT solvers

SMT solvers are the tools we use to solve SATs.
SAT problems is translate in propositional formula that is composed by boolean variables connected by boolean operators $(, \not\vee, \rightarrow)$.

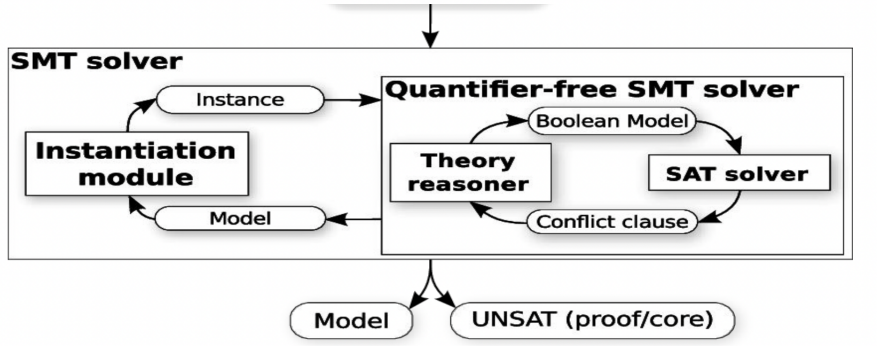
- if the formula is satisfiable, there is an assignment that makes the results true, otherwise the formula is *unsat*
- the solution to the formula are based on the truth table, the solution to which is exponential
- SAT is an NP-Complete problem

Z3 SAT/SMT Solver

Z3 SAT/SMT is a SAT solver developed by Microsoft

- an instance is passed to the quantifier-free SMT solver
- theory reasoner creates a boolean model, which is then tested by the SAT solver
- if there are conflict clauses, the model is rejected and the theory reasoner tries again

The SMT solver then outputs a model which can be further developed, or the expression is deemed UNSAT



Other SMT solvers are Yices by SRI, STP by Vijay Ganesh, CVC3 by NYU

A bit of history

- SAGE is a concolic executor developed at Microsoft Research which primarly targets bugs in file parsers (JPEG, DOCX, PPT etc.), uses generational search. SAGE is used on production software at MS (500 machines years, 3.4+ billion constrains, 100s of apps, 100s of bugs)
- KLEE is a symbolic executor for LLVM bitcode, uses $fork()$ to manage multiple states, uses a variety of search strategies, mainly random path + coverage guided
- Mayhem runs on binaries, uses BFS search and native execution, combines best of symbolic and concolic strategies, and automatically generates exploits when bugs are found
- Mergepoint extends Mayhen with a technique called veritesting, combining symbolic execution with static analysis. It provides a better balance of time between olver and executor, finding more bugs, and covers more of the program