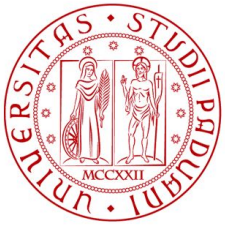
CyberSecurity: Principle and Practice

BSc Degree in Computer Science
2024-2025

Lesson 17: PLT & GOT

Prof. Mauro Conti
Department of Mathematics
University of Padua
conti@math.unipd.it
http://www.math.unipd.it/~conti/

Teaching Assistants
Tommaso Bianchi
tommaso.bianchi@phd.unipd.it.
Riccardo Preatoni
riccardo.preatoni@phd.unipd.it



UNIVERSITÀ DEGLI STUDI DI PADOVA

SPRITZ SECURITY & PRIVACY RESEARCH GROUP

DIPARTIMENTO MATEMATICA

Disclaimer

All information presented here has the only purpose of teaching how reverse engineering works.

Use your mad skillz only in CTFs or other situations in which you are legally allowed to do so.

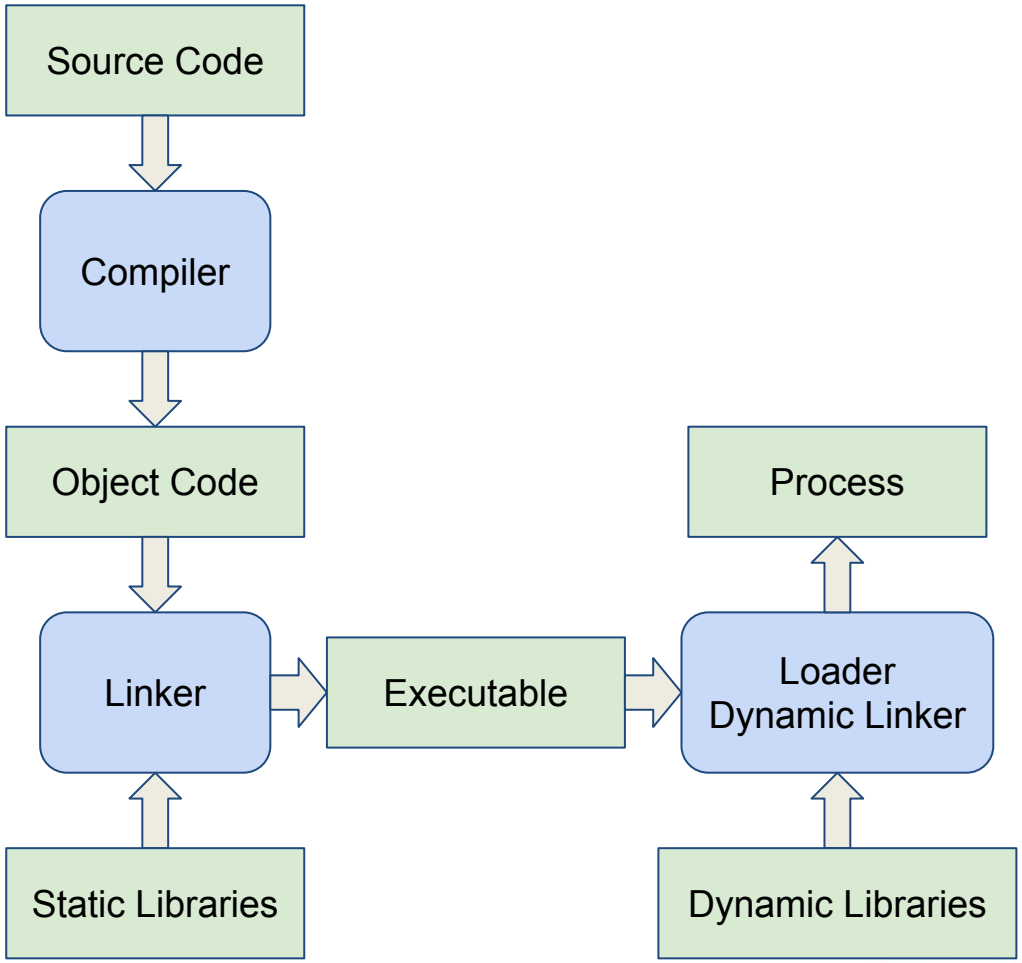
Do not hack the new Playstation. Or maybe do, but be prepared to get legal troubles 😊

ABUSING ELF DYNAMIC LINKING

With some vulnerabilities, you have the opportunity to write **arbitrary data** to **(almost) arbitrary memory addresses** (e.g., out-of-bounds array accesses).

One way to exploit this is by **abusing the internals of ELF dynamic linking**.

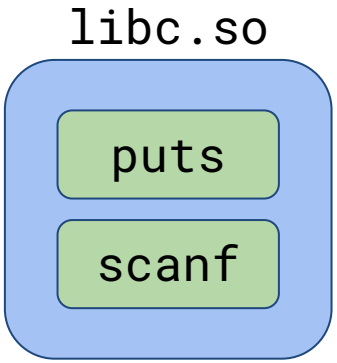
A PROGRAM'S LIFECYCLE



Global Offset Table (GOT)

Program imports (GOT)

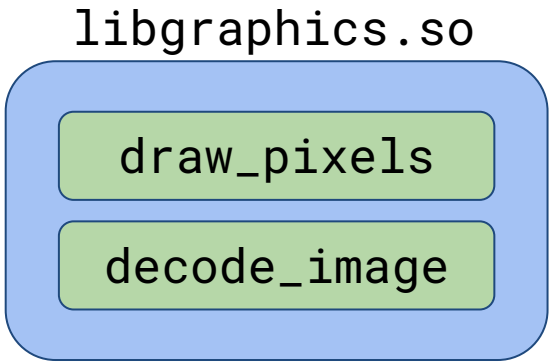
puts = ?
scanf = ?
decode_image = ?
draw_pixels = ?



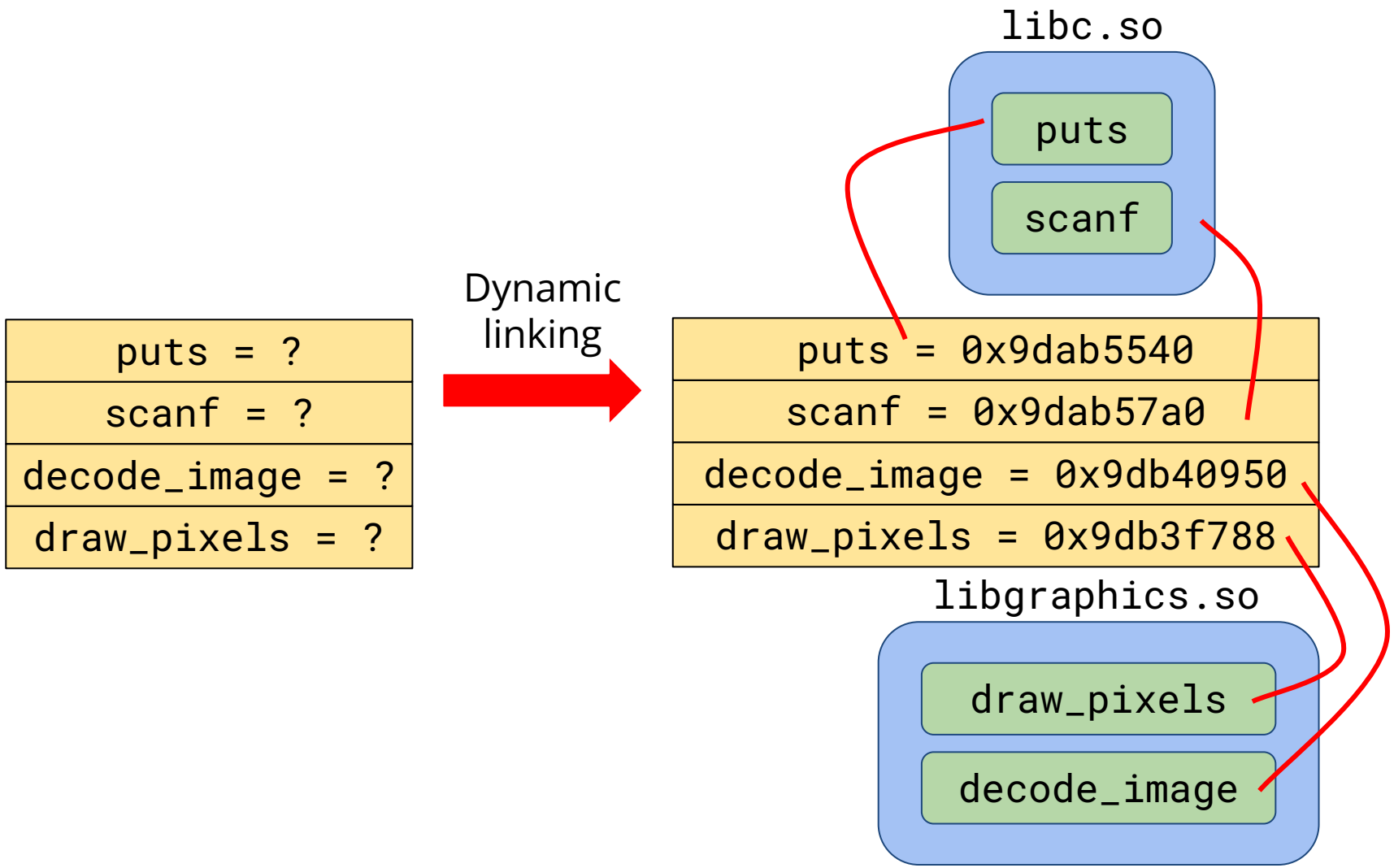
External Libraries

GOT enables compiled code (such as ELF) to run correctly

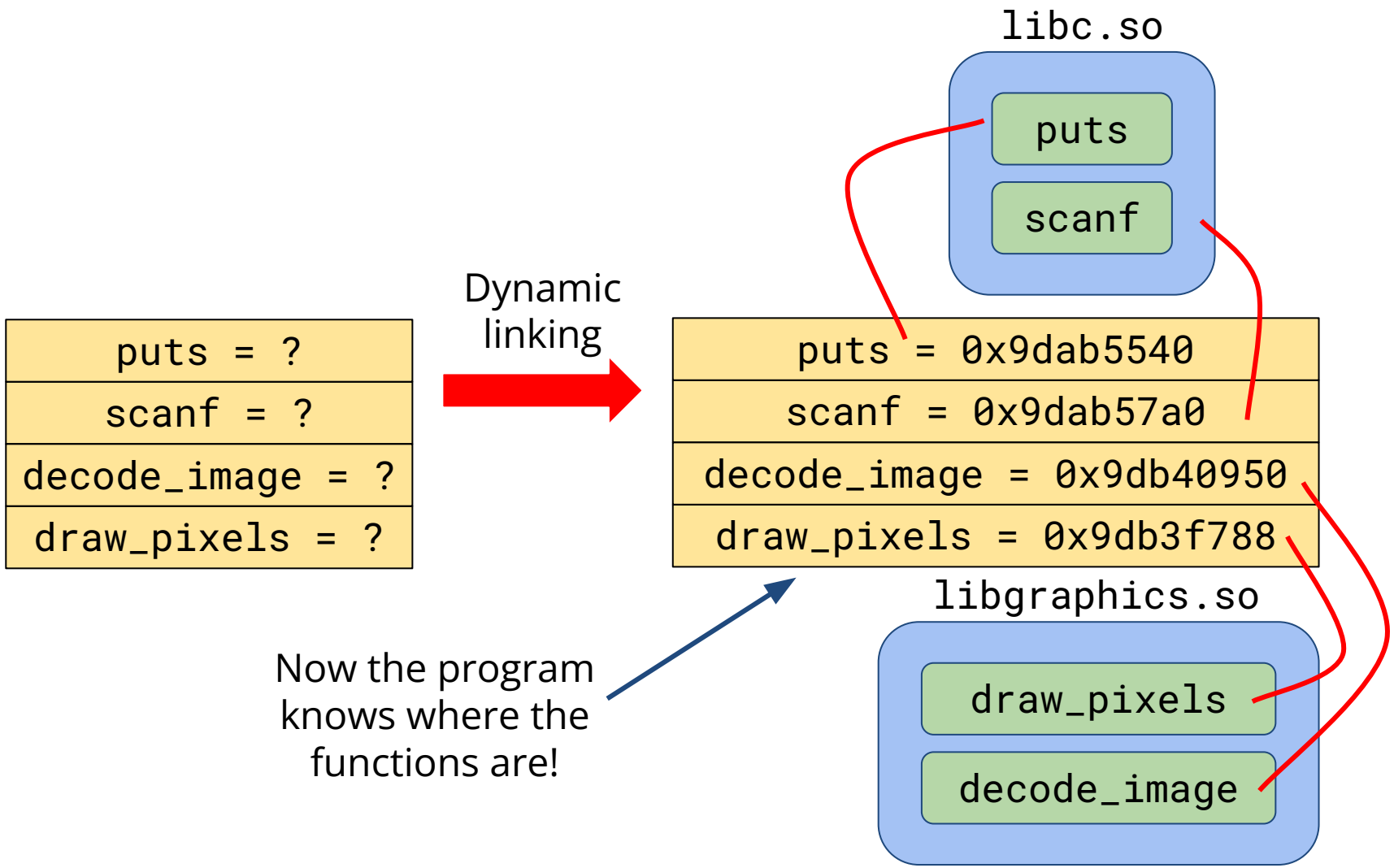
- **independently from actual memory addresses where loaded at runtime**



Global Offset Table (GOT)



Global Offset Table (GOT)



Global Offset Table (GOT) in IDA


```
.got.plt:0000000000202000 ; ===========================================================
.got.plt:0000000000202000
.got.plt:0000000000202000 ; Segment type: Pure data
.got.plt:0000000000202000 ; Segment permissions: Read/Write
.got.plt:0000000000202000 ; Segment alignment 'qword' can not be represented in assembly
.got.plt:0000000000202000 _got_plt        segment para public 'DATA' use64
.got.plt:0000000000202000                 assume cs:_got_plt
.got.plt:0000000000202000                 ;org 202000h
.got.plt:0000000000202000 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000202008 qword_202008    dq 0                  ; DATA XREF: sub_8A0□r
.got.plt:0000000000202010 qword_202010    dq 0                  ; DATA XREF: sub_8A0+6□r
.got.plt:0000000000202018 off_202018      dq offset recv        ; DATA XREF: _recv□r
.got.plt:0000000000202020 off_202020      dq offset _Z10uuid_parsePKcPh
.got.plt:0000000000202020                                       ; DATA XREF: uuid_parse(ch
.got.plt:0000000000202020                                       ; uuid_parse(char const*,u
.got.plt:0000000000202028 off_202028      dq offset write       ; DATA XREF: _write□r
.got.plt:0000000000202030 off_202030      dq offset strlen      ; DATA XREF: _strlen□r
.got.plt:0000000000202038 off_202038      dq offset __stack_chk_fail
.got.plt:0000000000202038                                       ; DATA XREF: ___stack_chk_
.got.pl                   f_202040        dq offset htons       ;
.got.pl                   f_202048        dq offset memset      ;
.got.pl                   f_202050        dq offset close       ;
.got.pl                   f_202058        dq offset memcpy      ;
.got.pl                   f_202060        dq offset inet_aton   ;                    n□r
.got.plt:0000000000202068 off_202068      dq offset perror      ; DATA XREF: _perror□r
.got.plt:0000000000202070 off_202070      dq offset strtoul     ; DATA XREF: _strtoul□r
.got.plt:0000000000202078 off_202078      dq offset connect     ; DATA XREF: _connect□r
.got.plt:0000000000202080 off_202080      dq offset isxdigit    ; DATA XREF: _isxdigit□r
.got.plt:0000000000202088 off_202088      dq offset socket      ; DATA XREF: _socket□r
.got.plt:0000000000202088 _got_plt        ends
```

Offset in the GOT to call write

Actual write function address

Procedure Linkage Table (PLT)

To reach the GOT table, we have another indirection, PLT

- Collection of ***trampolines*** **(one for each import)**
  - Program calls PLT entry
  - PLT entry `jmp`s through the GOT
- Why this extra indirection?
  - Lazy linking
  - Non-PIE (Position-independent- executable) dynamically-linked programs

More: https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html

Procedure Linkage Table (PLT)

GOT Hijacking

1. **Overwrite a GOT entry** via some memory corruption
2. Make the program call the corresponding PLT function
   - Will be dispatched through the GOT
3. ?!?
4. Profit, got PC control :)

FUNCTION REUSE EXAMPLE

Let's assume we had a chance for GOT hijacking before this code:

```
char buf[100];
scanf("%99s", buf);
puts(buf);
```

FUNCTION REUSE EXAMPLE

Let's assume we had a chance for GOT hijacking before this code:

```
char buf[100];
scanf("%99s", buf);
puts(buf);
```

What happens if we overwrite the **puts** GOT entry with the address of **system**?

FUNCTION REUSE EXAMPLE

Let's assume we had a chance for GOT hijacking before this code:

```
char buf[100];
scanf("%99s", buf);
puts(buf);
```

What happens if we overwrite the `puts` GOT entry with the address of `system`?

```
char buf[100];
scanf("%99s", buf);
system(buf);
```

Argument to `system()` is attacker-controlled!

Lazy Linking

- Problem: finding symbols is slow
  - We're resolving all symbols **at startup**
  - Slow startup times => user is unhappy!
- Observation: most symbols aren't actually used
  - (in a specific execution)
- Solution: **lazy linking**
  - Delay symbol resolution until actually used

Lazy Linking

What does this mean to an attacker?

- Need to **corrupt a GOT entry for a function**

If PLT finds the function's GOT entry empty:

- it will resolve its symbol and get the real function address
- Otherwise, it will directly call the function at the address specified in the GOT.

→ If we write in the GOT our malicious function's address, the program will call it! Profit :)

Note: It's better to overwrite the entry of a function that has already been called once.

RELOCATION READ-ONLY (RELRO)

- **Full RELRO**: whole GOT is read-only
  - \+ makes the **whole GOT read-only to avoid hijacking**
  - \- incompatible with lazy linking
- **Partial RELRO**: part of the GOT is readonly (part managing global variables---not our business here), but still functions have problems.
  - \+ **compatible with lazy linking**
  - \- hijacking still possible

Exercises

1) Can you spawn a shell and get the flag?
2) If you mess some bytes around, you might print the flag :)
3) This is a position-independent binary which gives you a module address, and a trivial write-what-where. Can you spawn a shell?

Questions? Feedback? Suggestions?