

Defenses

How can we defend against stack-based overflows?

- Write code w/o vulnerabilities (unrealistic)
- Use memory safe languages (if you can)
- Safer APIs (`strncpy`, `std::string`)
- Shadow stack (complicated, inefficient)
- Stack canaries / cookies
- ASLR, NX

Stack Cookies

```
> gcc -fstack-protector-all -o not-vuln vuln.c  
> ./not-vuln aaaa...  
*** stack smashing detected ***: ./not-vuln terminated
```

- Proposed by Cowan et al. in 1998 (StackGuard) and is a simple idea
 - Guard sensitive data, including the saved IP, with a copy of a secret value
 - Before returning, check the value against the original
 - If there is a difference, assume something bad has happened and terminate

Stack Cookies

main:

```
    push rbx
    sub rbp, 0x110
    mov rax, qword fs:0x28      ; load secret at fs:0x28
    mov qword [rsp+0x108], rax ; store secret to top of frame
    [...]
    mov rax, qword fs:0x28      ; load secret again
    cmp rax, qword [rsp+0x108] ; compare the stack secret
    jne .bad                    ; if not equal, don't return
    xor eax, eax
    add rsp, 0x110
    pop rbx
    ret                          ; can return with confidence
.bad:
    call __stack_chk_fail       ; print a scary message and exit
```

Stack Cookies

- Required properties
 - Large domain
 - Random
- Example of a great success story
 - One simple compiler flag
- Downsides
 - Introduces bloat, worse cache behavior
 - Incomplete coverage in popular implementations
 - Susceptible to information leaks

Outline

Assembly Review

Vulnerabilities I

Vulnerabilities II

Defenses & Evasion of Defenses

Malware Analysis

Format Strings

```
int printf(const char* format, ...);
```

- Family of functions for formatting string data
- The format string controls presentation
 - Contains a mixture of static data and variable placeholders
 - Placeholders have a mini-grammar

```
%<pos><flags><out-width><prec><in-width><conv>
```

- Variables passed as additional arguments

Format Strings

- Many formatting directives
 - %s – String data
 - %d – Decimal numbers
 - %x – Hexadecimal numbers
 - %f – Floating point numbers
 - %p – Pointers
 - %n – Number of bytes written
- If an attacker can control a format string, any location in memory can be overwritten
- All members of the family are vulnerable
 - fprintf, snprintf, vprintf, vsnprintf, ...

Format Strings

- %n directive is special
 - Others simply print a value from the stack
- Instead, %n writes the number of bytes printed so far to an address
 - This sounds useful for an attacker...
- Requirements
 1. Control the number of bytes written
 2. Control the destination address

Vulnerable Program

```
int main(int argc, char **argv)
{
    char buf[256];
    snprintf(buf, sizeof(buf), argv[1]);
    printf("buf = %s\n", buf);
    return 0;
}
```

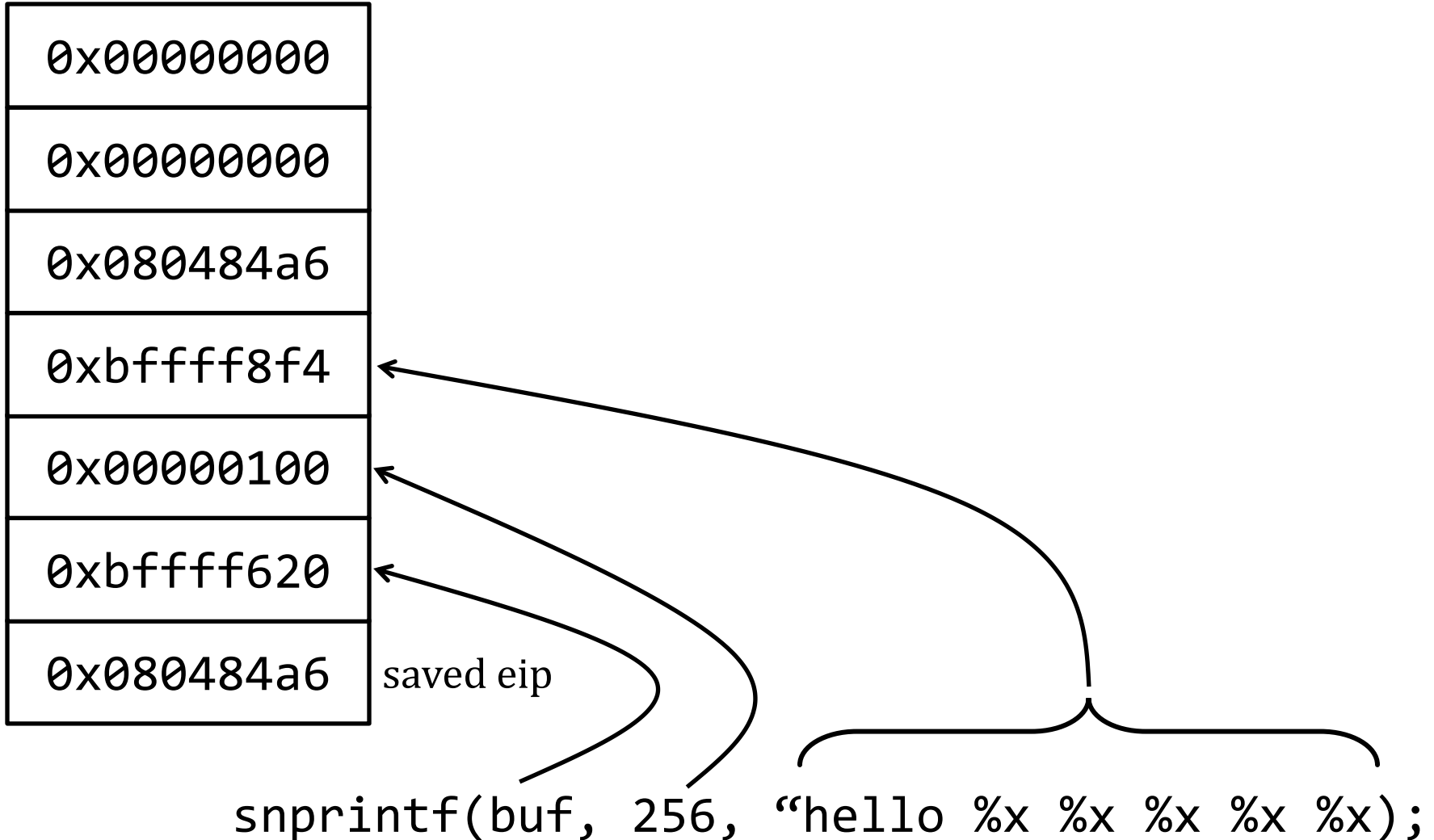
```
> ./vuln hello
```

```
buf = hello
```

```
> ./vuln 'hello %x %x %x %x %x'
```

```
buf = hello 0 f7dd7c60 f7deae20 ffffe808 f7ffe1a8
```

Vulnerable Program



Vulnerable Program

```
(gdb) b *0x080484a1
```

```
Breakpoint 1 at 0x080484a1
```

```
(gdb) r 'hello %x %x %x %x %x'
```

```
Starting program: /home/user/vuln 'hello %x %x %x %x %x'
```


```
Breakpoint 1, 0x080484a1 in main ()
```

```
(gdb) si
```

```
0x08048350 in snprintf@plt ()
```

```
(gdb) x/8wx $esp
```

```
0xbffff60c: 0x080484a6 0xbffff620 0x00000100 0xbffff8f4  
0xbffff61c: 0x00000000 0x00000000 0x00000000 0x00000000
```



Exploitation

```
(gdb) r 'AAAA %x %x %x %x %x'
```

```
Starting program: /home/user/vuln 'AAAA %x %x %x %x %x'
```

```
buf = AAAA 0 41414141 34203020 34313431 20313431
```

- By printing up the stack, we can find a known sequence of bytes
 - But, so far we can only print data ...

Exploitation

```
(gdb) r 'AAAA %x %n'
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0xb7e6fc60 in _IO_vfprintf_internal
```

```
(gdb) x/i $eip
```

```
=> 0xb7e6fc60:      mov     DWORD PTR [eax],ecx
```

```
(gdb) p/x $eax
```

```
$1 = 0x41414141
```

```
(gdb) p/x $ecx
```

```
$2 = 0x7
```

- We can use %n to write the number of bytes printed so far to an address we control

Exploitation

- Choose the address of data to overwrite
- Write that address to the stack
- Find that address on the stack
 - Print contents of stack using %x
 - Or, use gdb
- Use %n to write data to the address

Constructing a Value

```
(gdb) r '%032x'
```

```
buf = 0000000000000000000000000000000000000000000000000000000000000000
```



32 x 0

- To control the value to be written, we must print that number of bytes
 - Padding useful to increase number
 - E.g., %32x prints at least 32 characters

Demo
Format String
Corrupting Return Address

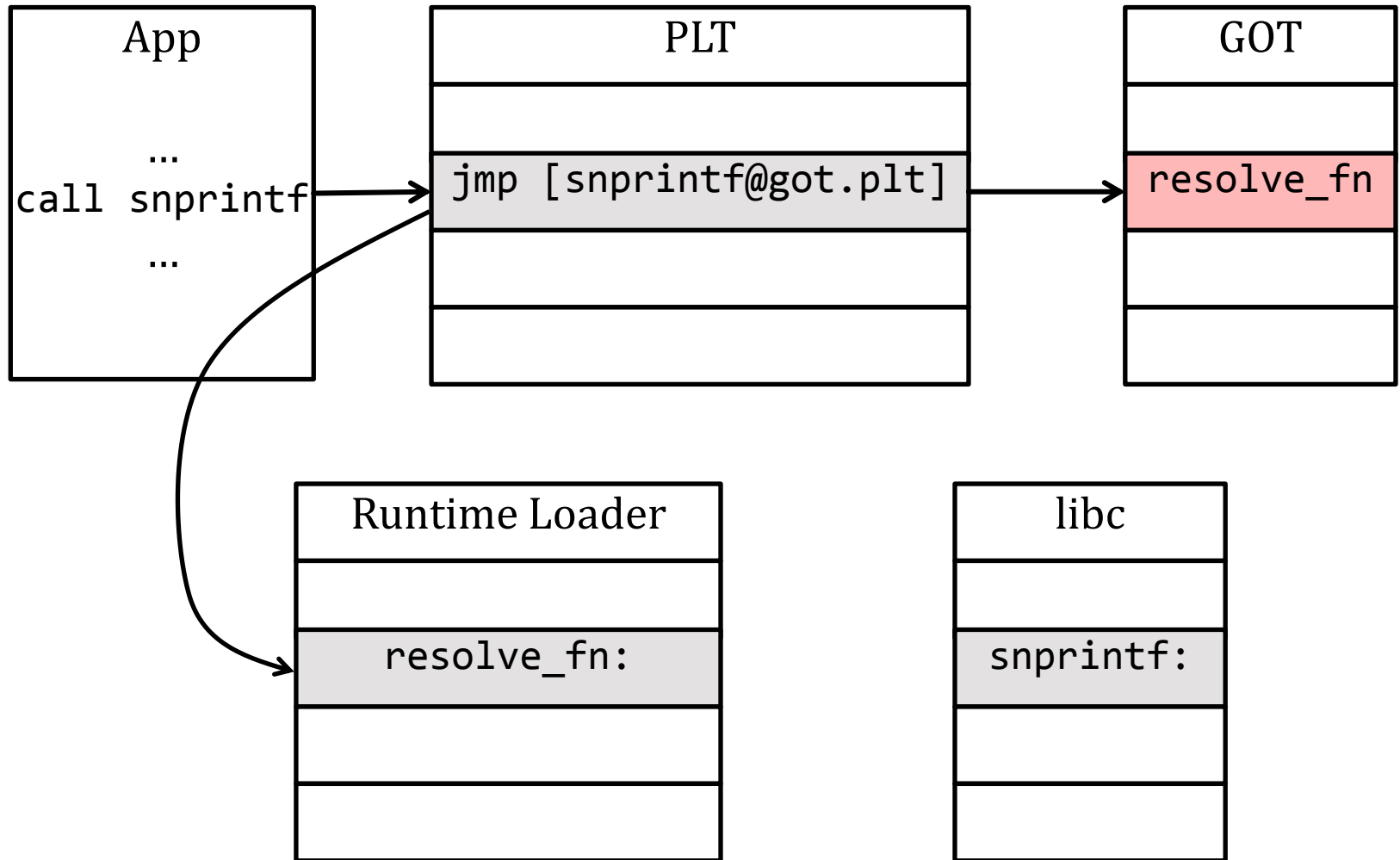
Function Pointers

- So far, we've considered one way of hijacking control flow:
 - Corrupting the return address on the stack
- But, any function pointer can be used to hijack execution
 - **PLT entries**
 - ctor, dtors
 - vtable entries
 - Tricking the heap manager using fake chunks

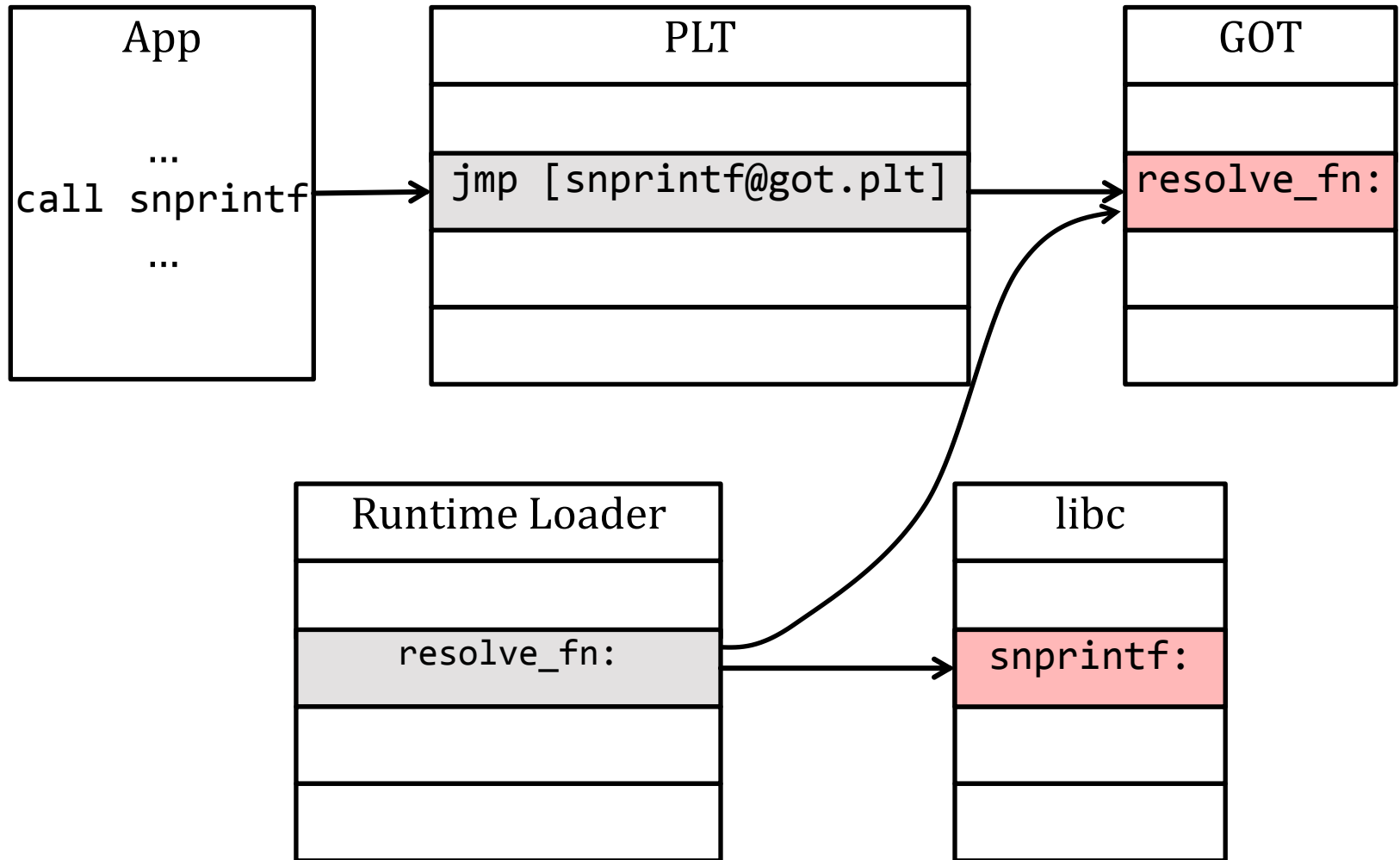
PLT

- Dynamically-linked executables reference external (library) functions
 - But, it isn't known at link time where those functions will be located
- Procedure Linkage Table (PLT)
 - Along with the Global Offset Table (GOT), used to implement runtime function resolution
 - Caches locations of external functions

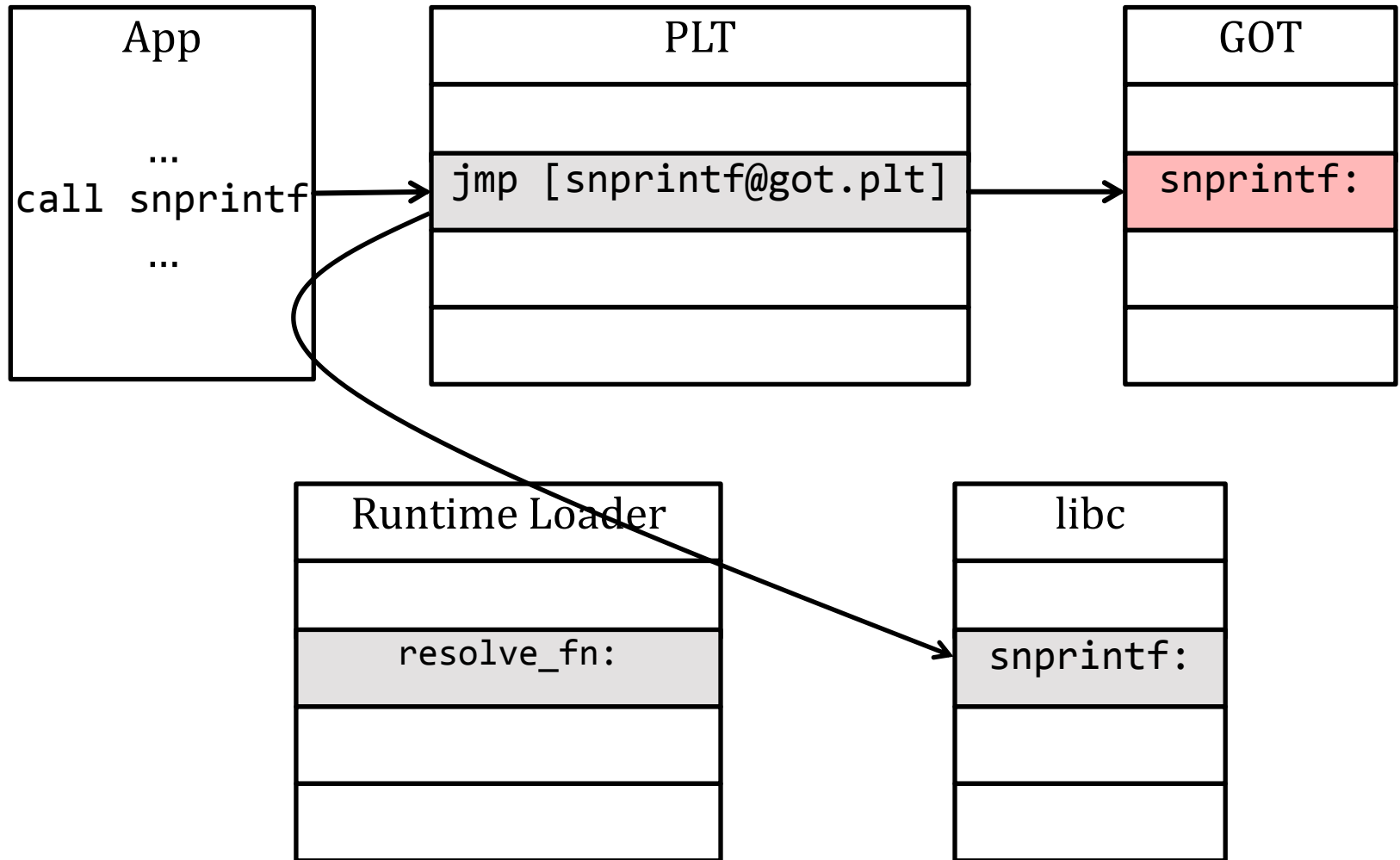
Function Resolution



Function Resolution



Function Resolution



PLT Entries

- PLT references writable function pointers in the GOT
 - These pointers are prime targets for hijacking execution
 - Often, PLT is also conveniently mapped at a fixed address ...
- Exploitation
 - Overwrite a GOT entry to point to attacker code
 - Coerce program into executing corresponding external function

Demo
Format String
Corrupting GOT Entry