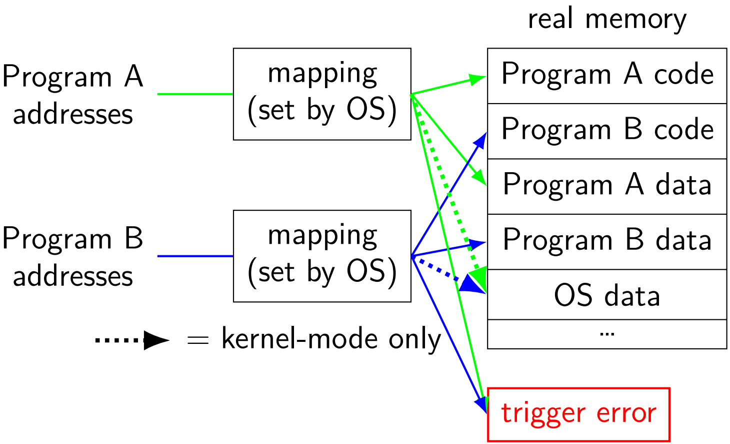
recall(?): virtual memory

illuision of *dedicated memory*



the mapping (set by OS)

program address range	read?	write?	real address
0x0000 --- 0x0FFF	no	no	---
0x1000 --- 0x1FFF	no	no	---
...			
0x40 0000 --- 0x40 0FFF	yes	no	0x...
0x40 1000 --- 0x40 1FFF	yes	no	0x...
0x40 2000 --- 0x40 2FFF	yes	no	0x...
...			
0x60 0000 --- 0x60 0FFF	yes	yes	0x...
0x60 1000 --- 0x60 1FFF	yes	yes	0x...
...			
0x7FFF FF00 0000 — 0x7FFF FF00 0FFF	yes	yes	0x...
0x7FFF FF00 1000 — 0x7FFF FF00 1FFF	yes	yes	0x...
...			

Virtual Memory

modern *hardware-supported* memory protection mechanism

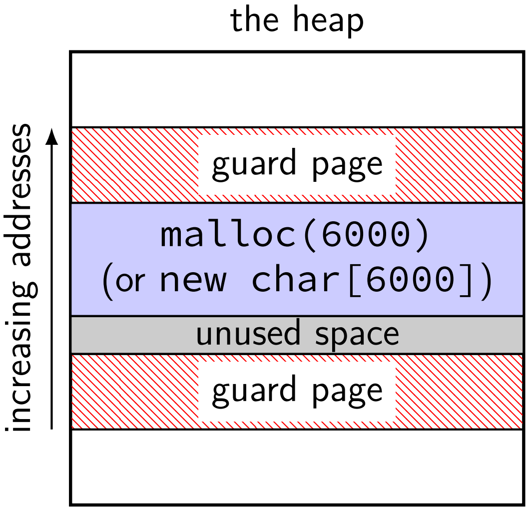
via *table*: OS decides *what memory program sees*

- whether it's read-only or not

granularity of *pages* — typically 4KB

not in table — segfault (OS gets control)

malloc/new guard pages



guard pages

deliberate holes

accessing — segfualt

call to OS to allocate (not very fast)

likely to ‘waste’ memory

- guard around object? minimum 4KB object

guard pages for malloc/new

can implement malloc/new by placing guard pages around allocations

- commonly done by real malloc/new's for *large allocations*

problem: minimum actual allocation 4KB

problem: substantially slower

example: "Electric Fence" allocator for Linux (early 1990s)

guard pages and arrays/structs

```
struct foo {
    char buffer[10000];
    /* can't really put guard page here */
    int *ptr;
};
```

C compiler expects buffer and ptr to be adjacent

can't add guard page without changing all code that accesses `struct foo`

similar problem with separating elements of arrays

exercise: guard page overhead

suppose heap allocations are:

- $100\,000$ objects of 100 bytes
- $1\,000$ objects of 1000 bytes
- 100 objects of approx. 10000 bytes

total allocation of approx 12 000 KB

assuming 4KB pages, estimate space overhead of using guard pages:

- for objects larger than 4096 bytes (1 page)
- for objects larger than 200 bytes
- for all objects

solution (greater than 4096 byte)

100 objects of approx. 10000 bytes

- need to pad to 12288 (3 x 4096) bytes
- $228\,800$ wasted bytes for $1\,000\,000$ bytes of allocations

$1\,000$ objects of approx. 1000 bytes

- need to pad to (4096) bytes
- $3\,049\,000$ wasted bytes for $1\,000\,000$ bytes of allocations

$100\,000$ objects of approx 100 bytes

- need to pad to (4096) bytes
- $39\,490\,00$ wasted bytes for $10\,000\,000$ bytes of allocations

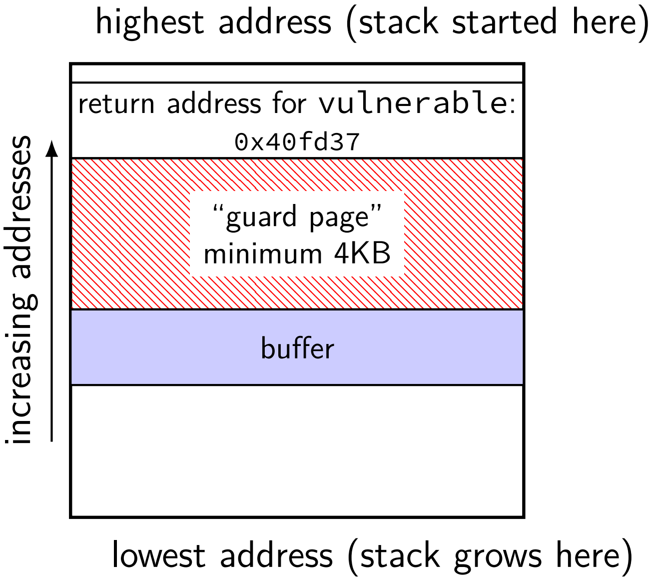
guard pages elsewhere?

could potentially add guard pages between big global variables

could potentaillly add guard pages after arrays on the stack

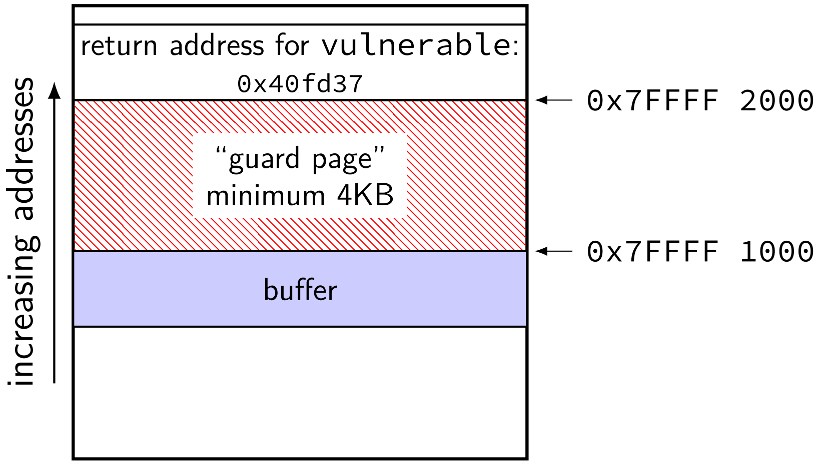
I don't know any systems that do this

stack canary alternative



stack canary alternative

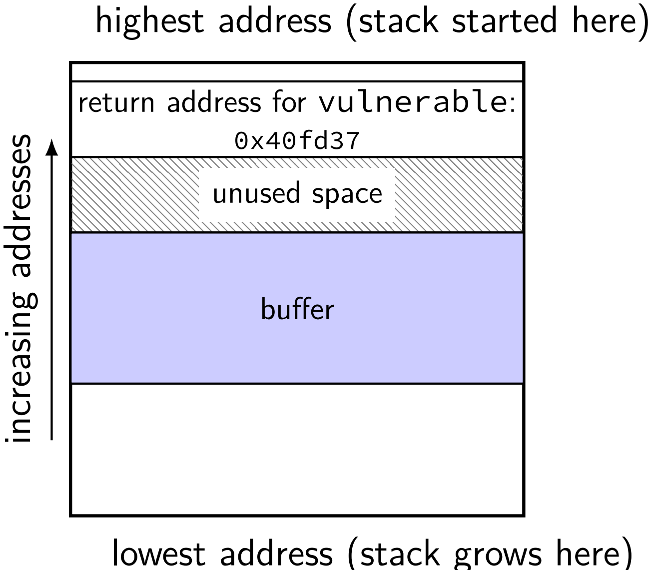
highest address (stack started here)



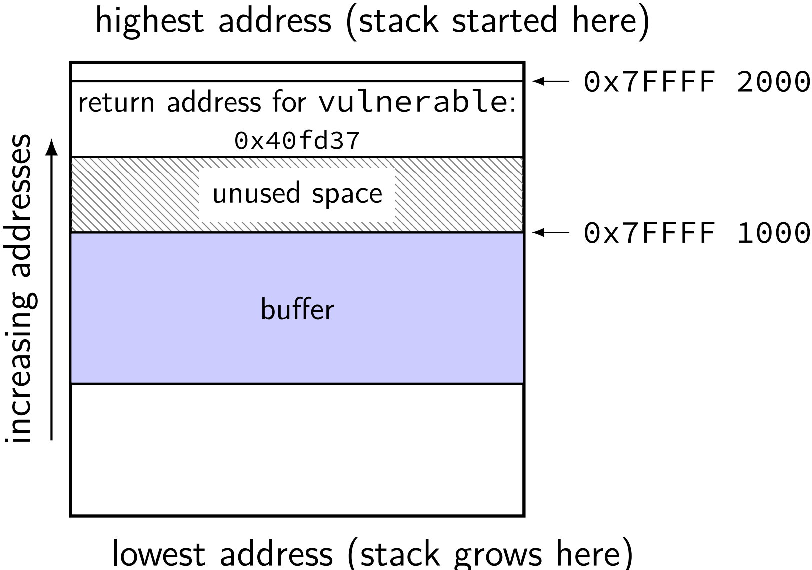
lowest address (stack grows here)

address	read	writ
0x7FFFF2000–0x7FFFF2FFF	yes	yes
0x7FFFF1000–0x7FFFF1FFF	no	no
0x7FFFF0000–0x7FFFF0FFF	yes	yes

stack canary alternative 2



stack canary alternative 2



address	read	write
0x7FFFF2000–0x7FFFF2FFF	yes	yes
0x7FFFF1000–0x7FFFF1FFF	yes	*no*
0x7FFFF0000–0x7FFFF0FFF	yes	yes

making things read-only

would really like to have things that shouldn't change be read-only

simple cases:

machine code

constants

separate sections

```
char *foo = "Hello";
char bar[] = "Hello";
```

turns into:

```
.data
bar:
    .string "Hello"
...
foo:
    .quad .LC0
.section .rodata.str1.1,"aMS",@progbits
# aMS = allocatable,mergeable,strings, @progbits = data
.LC0:
    .string "Hello"
```

separate segments (1)

```
LOAD off    0x0018000 vaddr 0x0018000 paddr 0x0018000 align 2
     filesz 0x0007458 memsz 0x0007458 flags r--
LOAD off    0x001ffd0 vaddr 0x0020fd0 paddr 0x0020fd0 align 2
     filesz 0x00012a8 memsz 0x0002570 flags rw-
```

separate segments (2)

compiler needs to separate constants/code/data into different segments

linker uses this info to make LOAD directives

- can mark some LOAD directives as read-only

need to add padding to make sure segments start at beginning of page

- one reason for rounding we saw in TRICKY

usually compiler writes *linker script* specifying order of sections + padding + how many LOAD directives

recall: function pointer targets

wanted to overwrite special pointer:

return addresses on stack

function pointers on in local variables

tables of function pointers used for inheritence

global offset table

can't realistically make first two read-only

read-only problems

global offset table and vtable entries produced at runtime

addresses of functions, etc. not chosen until program loaded

...or later with "lazy" linking

recall: filling in global offset tables as functions called

if we just set these as read-only, loading code will break

relocation data

addresses filled in by dynamic linker big target

- global offset table
- function pointers in vtables
- …

would like them to be read-only

…but they can't be read-only when initially loaded

RELRO

RELocation **R**ead-**O**nly

Linux option: make dynamic linker structures read-only after startup

partial RELRO: everything but GOT pointers to library functions

- notably includes C++ virtual function tables

full RELRO: everything including GOT pointers

- requires disabling "lazy binding" (filling in GOT as functions called)

appears as ELF program header entry

RELRO/non-lazy-binding in practice

linker/compiler options on Linux:

`-z relro`/`-z norerlo`: enable/disable relocation read-only

`-z now`: disable lazy binding (fill in whole GOT immediately)

in `objdump` (RELRO header; bit 3 of Dynamic Section FLAGS):

```
Program Headers:
...
   RELRO off    0x0000020f30 vaddr 0x0000021f30 paddr 0x00000
                filesz 0x00000010d0 memsz 0x00000010d0 flags r--
...
Dynamic Section:
...
   FLAGS                    0x0000000000000008
...
```

a thought on permissions

if we can set memory non-writeable

how about non-executable?

we never want to execute things on the stack anyways, right?

write XOR execute

many names:

- `W^X` (write XOR execute)
- DEP (Data Execution Prevention)
- NX bit (No-eXecute) (hardware support)
- XD bit (eXecute Disable) (hardware support)

mark writeable memory as executable

how will users insert their machine code?

- can only code in application + libraries
- a problem, right?

hardware support for write XOR execute

everywhere today

not historically common

early x86: execute implied by read

NX support added with x86-64 and around 2000 for x86-32

deliberate use of writeable code

"just-in-time" (JIT) compilers

- fast virtual machine/language implementations

some weird GCC features

older "signals" on Linux

- OS wrote machine code on stack for program to run

couldn't even disable executable stacks without breaking applications

why doesn't W xor X solve the problem?

W xor X is "almost free", keeps attacker from writing code?

problem: useful machine code is in program already

- just need to find writable function pointer

saw special case: arc injection

- use address of `system` function to replace `strlen`
- idea: find useful code already in application/library

turns out: almost always useful code

- trick: chaining together multiple pieces of machine code

backup slides