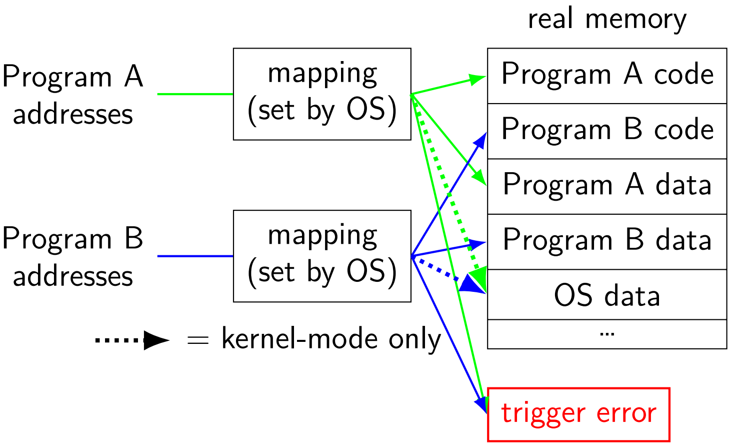
recall(?): virtual memory

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

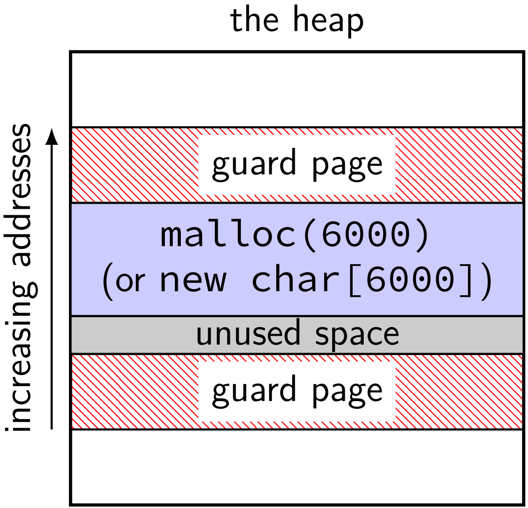
via *table*: OS decides *what memory program sees*

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

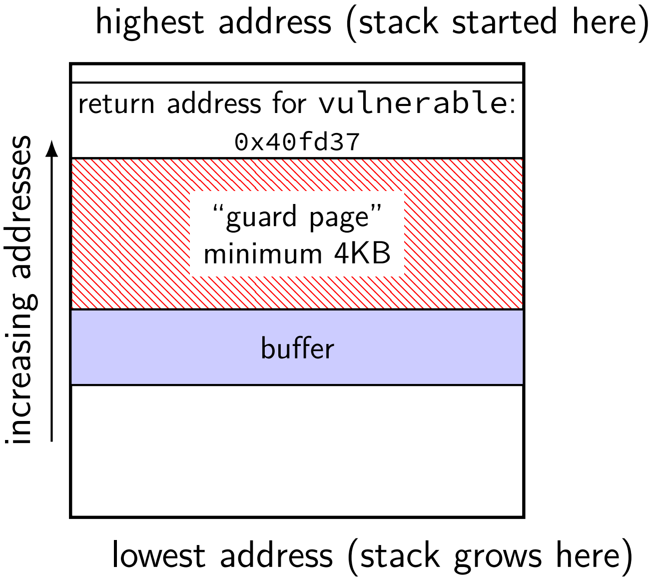
- commonly done by real malloc/new's for *large allocations*

problem: minimum actual allocation 4KB

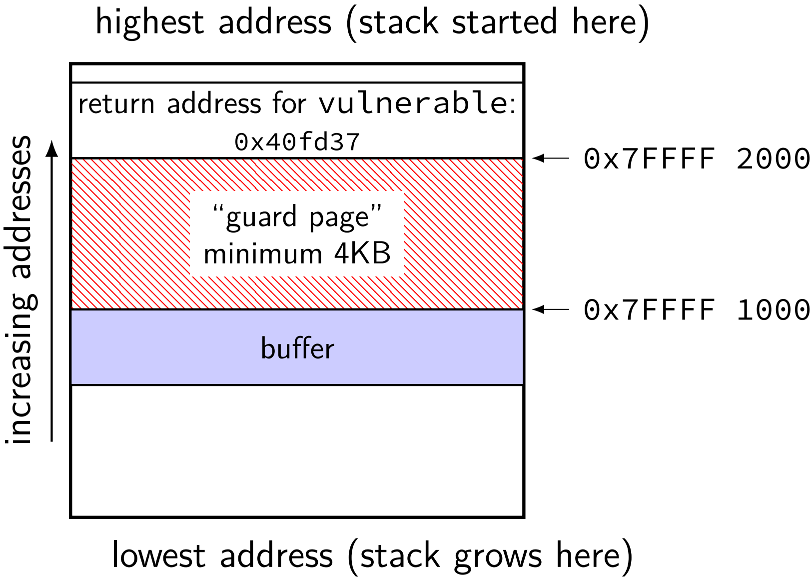
problem: substantially slower

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

stack canary alternative

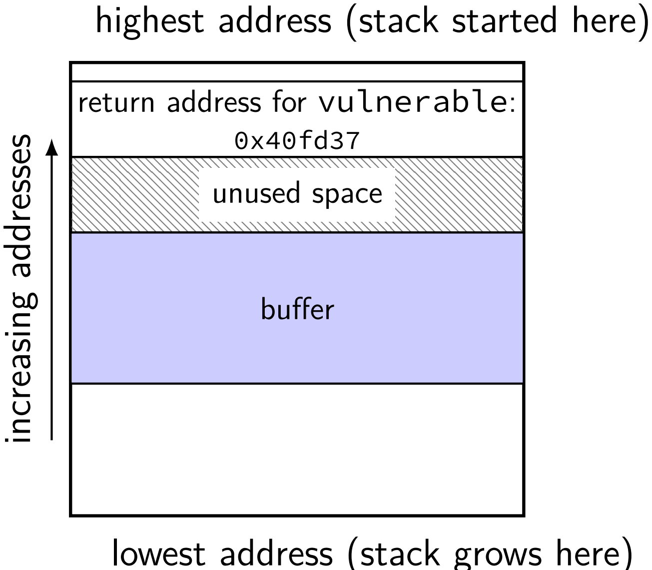


stack canary alternative

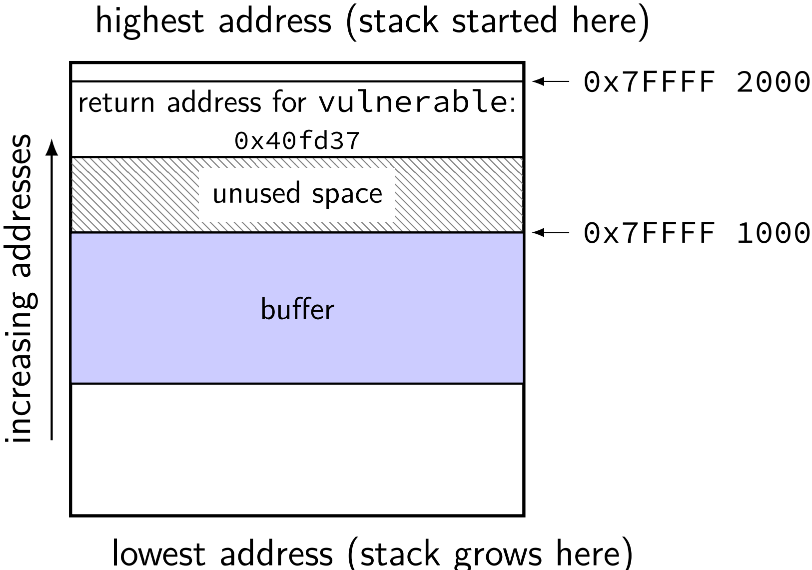


address	read	writ
0x7FFFF2000–0x7FFFF2FFF	yes	yes
0x7FFFF1000–0x7FFFF1FFF	no	no
0x7FFFF0000–0x7FFFF0FFF	yes	yes

stack canary alternative 2



stack canary alternative 2



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

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

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

last two: need to change infrequently

idea: make read-only

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 those pointers

- requires disabling "lazy" linking
- (could do without disabling — but slower (how much?) startup)

appears as ELF program header entry

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

- happened to find useful code in existing application/library

turns out: almost always useful code

backup slides