stopping stack smashing?

how can you stop stack smashing?

stopping stack smashing?

how can you stop stack smashing?

stop overrun — bounds-checking

stop return to attacker code

stop execution of attacker code

exploit mitigations

idea: turn vulnerablity to something less bad

e.g. crash instead of machine code execution

many of these targetted at buffer overflows

mitigation agenda

we will look briefly at one mitigation — stack canaries

then look at exploits that don't care about it

then look at more flexible mitigations

then look at more flexible exploits

mitigation priorities

effective? does it actually stop the attacker?

fast? how much does it hurt performance?

generic? does it require a recompile? rewriting software?

address space layout randomization (ASLR)

assume: addresses don't leak

choose *random* addresses each time

for *everything*, not just the stack

enough possibilities that attacker won't "get lucky"

should prevent exploits — can't write GOT/shellcode location

Linux stack randomization (x86-64)

1. choose random number between `0` and `0x3F FFFF`

2. stack starts at `0x7FFF FFFF FFFF` - *random number* $\times$ `0x1000`

   randomization disabled? *random number* $= 0$

16 GB range!

Linux stack randomization (x86-64)

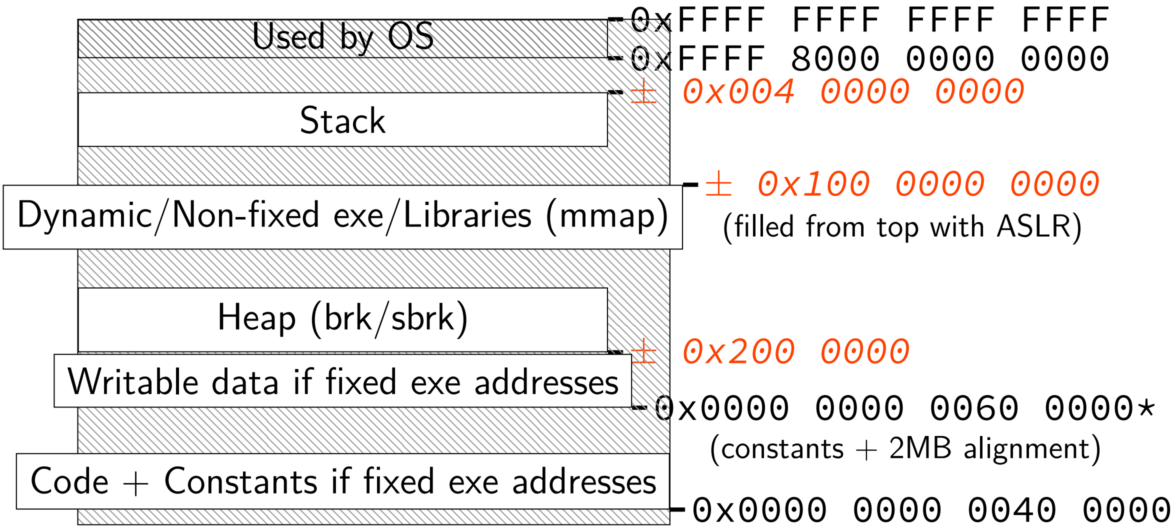
1. choose random number between `0` and *`0x3F FFFF`*

2. stack starts at `0x7FFF FFFF FFFF` - *random number* $\times$ `0x1000`

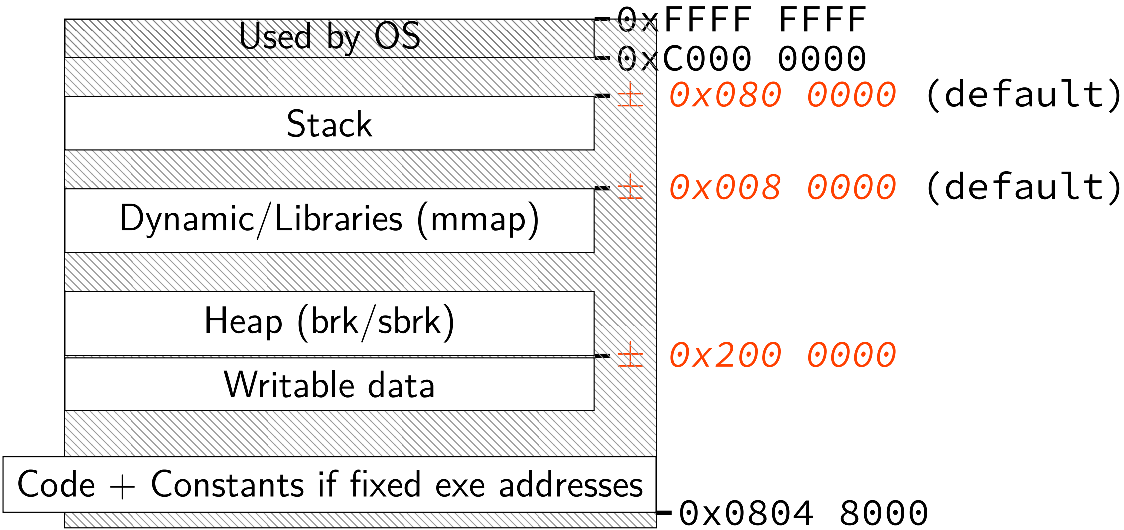
    randomization disabled? *random number* $= 0$

16 GB range!

program memory (x86-64 Linux; ASLR)



program memory (x86-32 Linux; ASLR)



how much guessing?

gaps change by multiples of page (4K)

- lower 12 bits are *fixed*

64-bit: *huge* ranges — need millions of guesses

- about *30 randomized bits* in addresses

32-bit: *smaller* ranges — hundreds of guesses

- only about *8 randomized bits* in addresses
- why? only 4 GB to work with!
- can be configured higher — but larger gaps

why do we get multiple guesses?

why do we get multiple guesses?

wrong guess might not crash

wrong guess might not crash whole application

- e.g. server that uses multiple processes

local programs we can repeatedly run

servers that are automatically restarted

dependencies between segments (1)

```
$ objdump -x foo.exe
...
LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x000
     filesz 0x0000000000000620 memsz 0x0000000000000620 flags r--
LOAD off    0x0000000000001000 vaddr 0x0000000000001000 paddr 0x000
     filesz 0x0000000000000205 memsz 0x0000000000000205 flags r-x
LOAD off    0x0000000000002000 vaddr 0x0000000000002000 paddr 0x000
     filesz 0x0000000000000150 memsz 0x0000000000000150 flags r--
LOAD off    0x0000000000002db8 vaddr 0x0000000000003db8 paddr 0x000
     filesz 0x000000000000025c memsz 0x0000000000000260 flags rw-
```

4 seperately loaded segments: can we choose random addresses for each?

dependencies between segments (2)

```
0000000000001050 <__printf_chk@plt>:
    1050:       f3 0f 1e fa             endbr64
    1054:       f2 ff 25 75 2f 00 00    bnd jmpq *0x2f75(%rip)
    105b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

dependency from 2nd LOAD (0x1000-0x1205) to 4th LOAD (0x3db8-0x4018)

uses relative addressing rather than linker filling in address

dependencies between segments (3)

```
0000000000001060 <main>:
    1060:       f3 0f 1e fa             endbr64
    1064:       50                      push   %rax
    1065:       8b 15 a5 2f 00 00       mov    0x2fa5(%rip),%edx
# 4010 <global>
    106b:       48 8d 35 92 0f 00 00    lea    0xf92(%rip),%rsi
# 2004 <_IO_stdin_used+0x4>
    1072:       31 c0                   xor    %eax,%eax
    1074:       bf 01 00 00 00          mov    $0x1,%edi
    1079:       e8 d2 ff ff ff          callq  1050 <__printf_chk@p
```

dependency from 2nd LOAD (0x1000-0x1205) to 3rd LOAD (0x2000-0x2150)

uses relative addressing rather than linker filling in address

why is this done?

Linux made a choice:
no editing code when loading programs, libraries

allows same code to be loaded in multiple processes

danger of leaking pointers

any stack pointer? know everything on the stack!

any pointer within executable? know everything in the executable!

any pointer to a particular library? know everything in library!

exericse: using a leak (1)

```
class Foo {
    virtual const char *bar() { ... }
};
...
Foo *f = new Foo;
printf("%s\n", f);
```

Part 1: What address is most likely leaked by the above?

A. the location of the Foo object allocated on the heap
B. the location of the first entry in Foo's VTable"
C. the location of the first instruction of Foo::Foo() (Foo's compiler-generated constructor)"
D. the location of the stack pointer

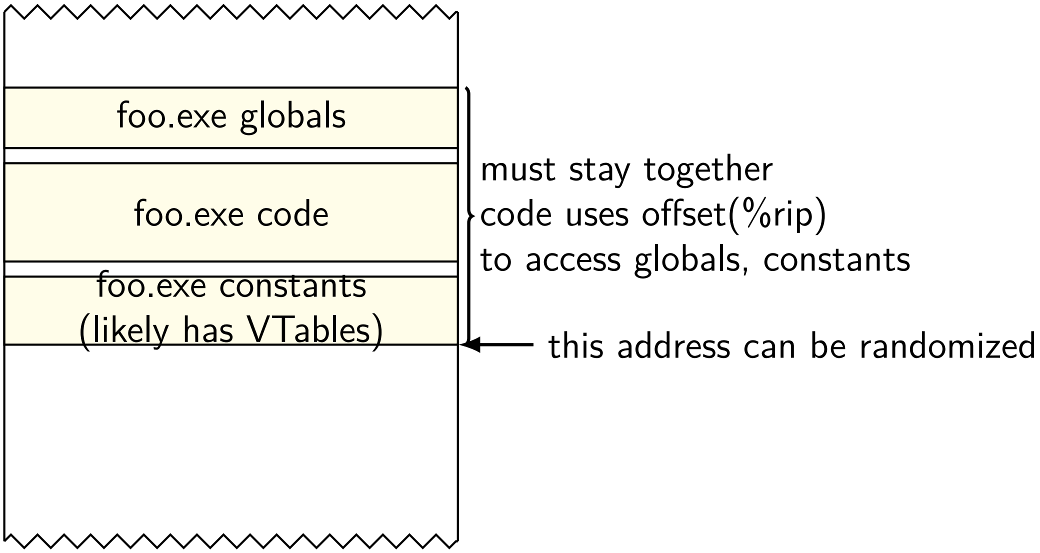
exercise: using a leak (2)

```
class Foo {
    virtual const char *bar() { ... }
};
...
Foo *f = new Foo;
char *p = new char[1024];
printf("%s\n", f);
```

if leaked value was 0x822003 and in a debugger (with **different randomization**):

- stack pointer was 0x7ffff000
- Foo::bar's address was 0x400000
- f's address was 0x900000
- f's Vtable's address was 0x403000
- a "gadget" address from the main executable was 0x401034
- a "gadget" address from the C library was 0x2aaaa40034
- p's address was 0x901000

exes, libraries stay together



relocating: Windows

Windows will *edit code* to relocate

- not everything uses a GOT-like lookup table

typically one fixed location per program/library **per boot**

- same address used across all instances of program/library
- still allows sharing memory

fixup once per program/library per boot

- before ASLR: code could be pre-relocated

Windows + Visual Studio had 'full' ASLR by default since 2010

Windows ASLR limitation

same address in all programs — not very useful against local exploits

PIC: Linux, OS X

Linux, OS X: position-independent code

allows libraries code pages to be shared

...even if loaded at different addresses

avoids per-boot randomization of Windows, but...

exercise: avoiding absolute addresses

```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
returnOne:
        movl    $1, %eax
        ret
returnTwo:
        movl    $2, %eax
defaultCase:
        ret
```

```
lookupTable:
    .quad returnOne
    .quad returnTwo
    .quad returnOne
    .quad returnTwo
    .quad returnOne
    .quad returnOne
```

exercise: rewrite this without absolute addresses

but fast

PIE jump-table

```
foo:
  movl   $3, %eax
  cmpq   $5, %rdi
  ja     retDefault
  leaq   jumpTable(%rip),%rax
  movslq (%rax,%rdi,4),%rdx
  addq   %rdx, %rax
  jmp   *%rax
returnTwo:
  movl  $2, %eax
  ret
returnOne:
  movl  $1, %eax
defaultCase:
  ret
```

```
  .section      .rodata
jumpTable:
  .long returnOne-jumpTable
  .long returnTwo-jumpTable
  .long returnOne-jumpTable
  .long returnTwo-jumpTable
  .long returnOne-jumpTable
  .long returnOne-jumpTable
```

PIE jump-table

```
foo:
  movl   $3, %eax
  cmpq   $5, %rdi
  ja     retDefault
  leaq   jumpTable(%rip),%rax
  movslq (%rax,%rdi,4),%rdx
  addq   %rdx, %rax
  jmp   *%rax
returnTwo:
  movl  $2, %eax
  ret
returnOne:
  movl  $1, %eax
defaultCase:
  ret
```

```
  .section      .rodata
jumpTable:
  .long returnOne-jumpTable
  .long returnTwo-jumpTable
  .long returnOne-jumpTable
  .long returnTwo-jumpTable
  .long returnOne-jumpTable
  .long returnOne-jumpTable
```

PIE jump-table

```
00000000000007ab <foo>:
b8 03 00 00 00          mov    $0x3,%eax
48 83 ff 05             cmp    $0x5,%rdi
77 1b                   ja     7d0 <foo+0x25>
48 8d 05 ab 00 00 00    lea    0xab(%rip),%rax       # 868
48 63 14 b8             movslq (%rax,%rdi,4),%rdx
48 01 d0                add    %rdx,%rax
ff e0                   jmpq   *%rax
b8 02 00 00 00          mov    $0x2,%eax
c3                      retq
b8 01 00 00 00          mov    $0x1,%eax
c3                      retq
...
@ 868: -156 /* offset */
@ 870: -162
...
```

PIE jump-table

```
00000000000007ab <foo>:
b8 03 00 00 00          mov    $0x3,%eax
48 83 ff 05             cmp    $0x5,%rdi
77 1b                   ja     7d0 <foo+0x25>
48 8d 05 ab 00 00 00    lea    0xab(%rip),%rax         # 868
48 63 14 b8             movslq (%rax,%rdi,4),%rdx
48 01 d0                add    %rdx,%rax
ff e0                   jmpq   *%rax
b8 02 00 00 00          mov    $0x2,%eax
c3                      retq
b8 01 00 00 00          mov    $0x1,%eax
c3                      retq
...
@ 868: -156 /* offset */
@ 870: -162
...
```

PIE jump-table

```
0000000000007ab <foo>:
b8 03 00 00 00           mov    $0x3,%eax
48 83 ff 05              cmp    $0x5,%rdi
77 1b                    ja     7d0 <foo+0x25>
48 8d 05 ab 00 00 00     lea    0xab(%rip),%rax        # 868
48 63 14 b8              movslq (%rax,%rdi,4),%rdx
48 01 d0                 add    %rdx,%rax
ff e0                    jmpq   *%rax
b8 02 00 00 00           mov    $0x2,%eax
c3                       retq
b8 01 00 00 00           mov    $0x1,%eax
c3                       retq
...
@ 868: -156 /* offset */
@ 870: -162
...
```

added cost

replace `jmp *jumpTable(,%rdi,8)`

with:

`lea` (get table address — with relative offset)

`movslq` (do table lookup of offset)

`add` (add to base)

`jmp` (to computed base)

32-bit x86 is worse

no relative addressing for `mov`, `lea`, ...

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00     jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

32-bit x86 is worse

no relative addressing for `mov`, `lea`, …

even changes “stubs” for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00    jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

32-bit x86 is worse

no relative addressing for `mov`, `lea`, ...

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00     jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

PIE

position-independent executables (PIE)

- no hardcoded addresses

alternative: *edit code (not global offset table) at load time*

- Windows solution

GCC: `-pie -fPIE`

- `-pie` is linking option
- `-fPIE` is compilation option
- related option: `-fPIC` (position independent code)
  - used to compile runtime-loaded libraries

hard-coded addresses? (64-bit)

```c
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
        /* code for defaultCase, returnOne,
        ...
        .section        .rodata
lookupTable: /* read-only pointers: */
        .quad   returnOne
        .quad   returnTwo
        .quad   returnOne
        .quad   returnTwo
        .quad   returnOne
        .quad   returnOne
```

hard-coded addresses? (64-bit)

```
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo>:
b8 03 00 00 00      mov    $0x3,%eax
48 83 ff 05         cmp    $0x5,%rdi
        /* jump to defaultCase: */
77 12               ja     0x40058d
        /* lookup table jump: */
ff 24 fd
18 06 40 00         jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588 /* returnOne */
@ 400620: 0x400582 /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

hard-coded addresses? (64-bit)

```c
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

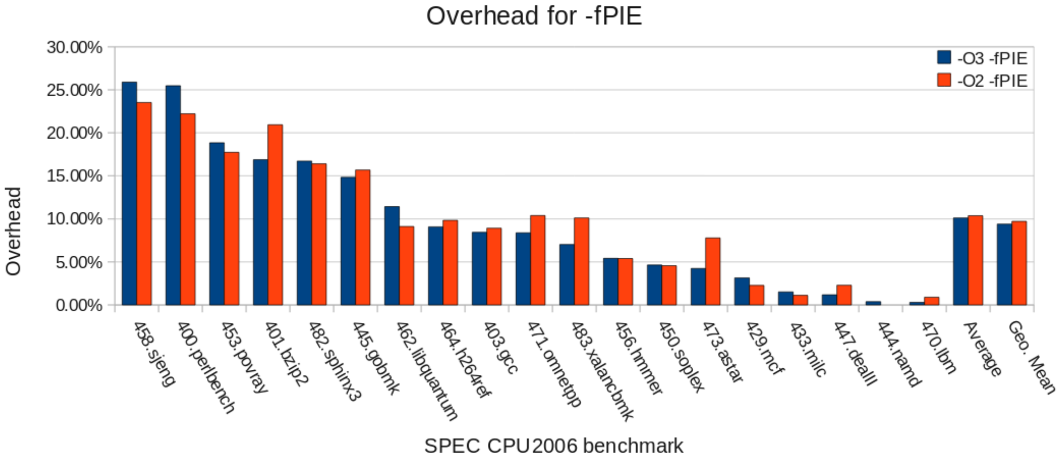
```
400570 <foo>:
b8 03 00 00 00     mov    $0x3,%eax
48 83 ff 05        cmp    $0x5,%rdi
       /* jump to defaultCase: */
77 12             ja     0x40058d
       /* lookup table jump: */
ff 24 fd
18 06 40 00       jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588 /* returnOne */
@ 400620: 0x400582 /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

hard-coded addresses? (64-bit)

```
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo>:
b8 03 00 00 00      mov    $0x3,%eax
48 83 ff 05         cmp    $0x5,%rdi
           /* jump to defaultCase: */
77 12              ja     0x40058d
           /* lookup table jump: */
ff 24 fd
18 06 40 00        jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588 /* returnOne */
@ 400620: 0x400582 /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

position independence cost (32-bit)



Payer, "Too much PIE is bad for performance", ETH Zurich Tech Report

position independence cost: Linux

geometric mean of SPECcpu2006 benchmarks on x86 Linux

with particular version of GCC, etc., etc.

64-bit: 2-3% (???)

"preliminary result"; couldn't find reliable published data

32-bit: 9-10%

depends on compiler, ...

position independence: deployment

common for a very long time in dynamic libraries

default for all executables in...

Microsoft Visual Studio 2010 and later
- `DYNAMICBASE` linker option

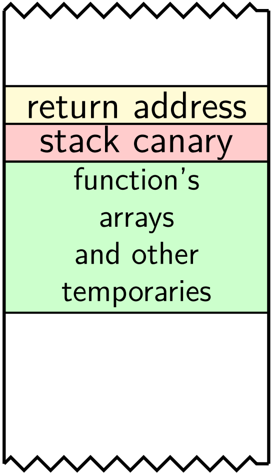
OS since 10.7 (2011)

Fedora 23 (2015) and Red Hat Enterprise Linux 8 (2019) and later
- default for "sensitive" programs earlier

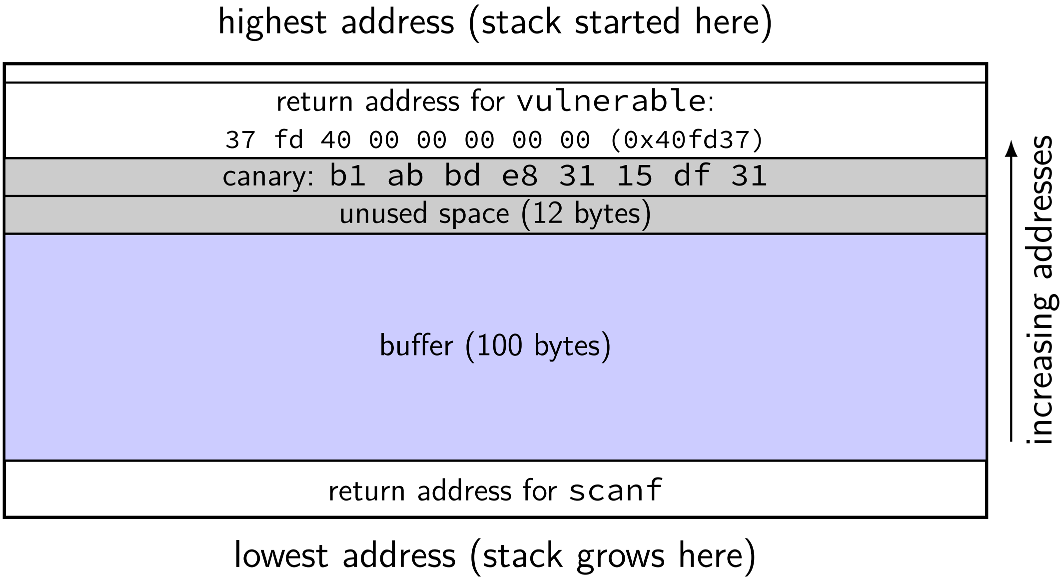
Ubuntu 16.10 (2016) and later (for 64-bit), 17.10 (2017) and later (for 32-bit)
- default for "sensitive" programs earlier

compiler generated code

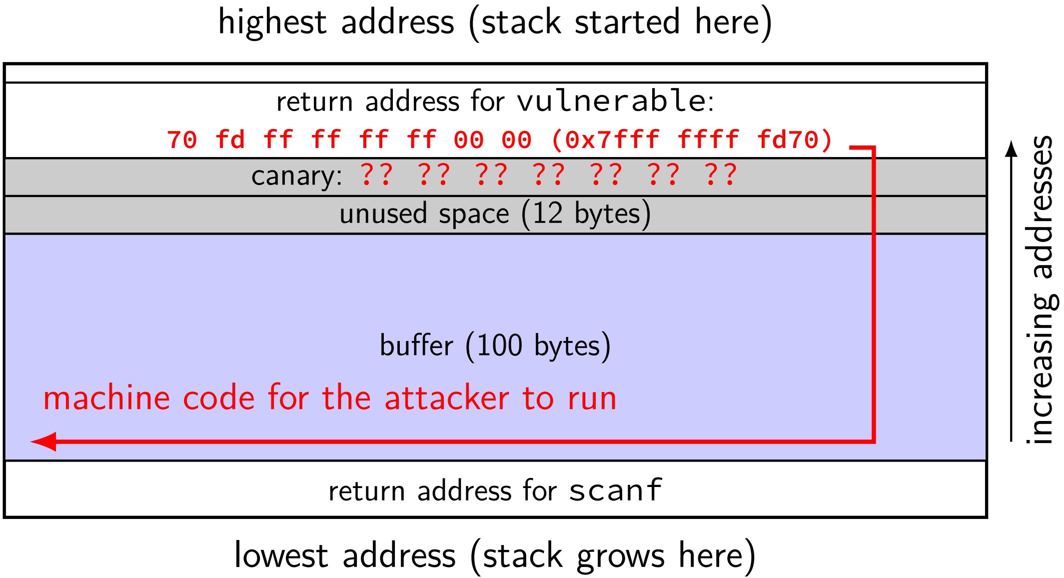
```
    pushq %rbx
    sub $0x20,%rsp
/* copy value from thread-local storage */
    mov $0x28,%ebx
    mov %fs:(%rbx),%rax
/* onto the stack */
    mov %rax,0x18(%rsp)
/* clear register holding value */
    xor %eax, %eax
    ...
    ...
/* copy value back from stack */
    mov 0x18(%rsp),%rax
/* xor to compare */
    xor %fs:(%rbx),%rax
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    ret
call_stack_chk_fail:
```



stack canary



stack canary



stack canary hopes

overwrite return address $\implies$ overwrite canary

canary is secret

good choices of canary

random — guessing should not be practical

- not always — sometimes static or only 2^{15} possible

GNU libc: canary contains:

leading `\0` (string terminator)

- `printf %s` won't print it
- copying a C-style string won't write it

a newline

- read line functions can't input it

`\xFF`

- hard to input?

stack canaries implementation

"StackGuard" — 1998 paper proposing strategy

GCC: command-line options

- `-fstack-protector`
- `-fstack-protector-strong`
- `-fstack-protector-all`
- one of these often default
- three differ in how many functions are 'protected'

Microsoft C/C++ compiler: `/GS`

- on by default

stack canary overheads

less than 1% runtime if added to “risky” functions

- functions with character arrays, etc.

large overhead if added to all functions

- StackGuard paper: 5–20%?

similar space overheads

(for typical applications)

- could be much worse: tons of ‘risky’ function calls

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: doesn't protect against many ways of exploiting buffer overflows

con: vulnerable to information leaks

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: *doesn't protect against many ways of exploiting buffer overflows*

con: vulnerable to information leaks

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: doesn't protect against many ways of exploiting buffer overflows

con: *vulnerable to information leaks*

stack canary summary

stack canary — simplest of many *mitigations*

key idea: detect corruption of return address

assumption: if return address changed, so is adjacent token

assumption: attacker can't learn true value of token

- often possible with memory bug

later: workarounds to break these assumptions

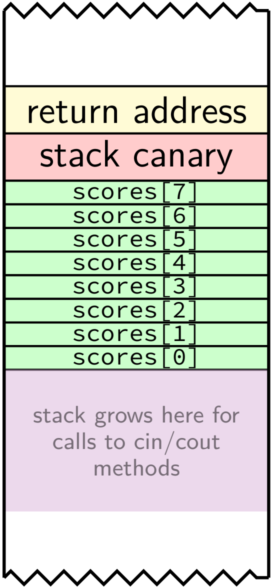
stack canary hopes

overwrite return address $\implies$ *overwrite canary*

canary is secret

non-contiguous overwrites

```
void vulnerable() {
  int scores[8]; bool done = false;
  while (!done) {
    cout << "Edit␣which␣score?␣(0␣to␣7)␣";
    int i;
    cin >> i;
    /* Oops!
       sizeof(scores) is 8 * sizeof(int) */
    if (i < 0 || i >= sizeof(scores))
      continue;
    cout << "Set␣to␣what␣value?" << endl;
    cin >> scores[i];
    ...
  }
  ...
}
```

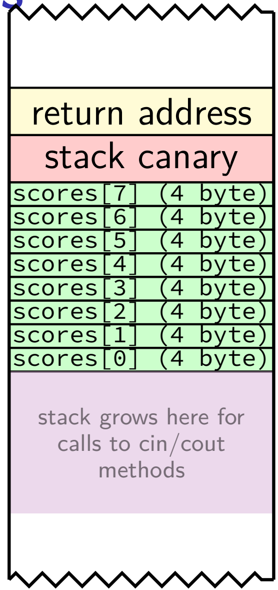


exercise: non-contiguous overwrites

```
void vulnerable() {
  int scores[8]; bool done = false;
  while (!done) {
    cout << "Edit␣which␣score?␣(0␣to␣7)␣";
    int i;
    cin >> i;
    /* Oops!
       sizeof(scores) is 4 * sizeof(int) */
    if (i < 0 || i >= sizeof(scores))
      continue;
    cout << "Set␣to␣what␣value?" << endl;
    cin >> scores[i];
```

exercise: to set return address to 0x123456789,
set what scores to what values?

```
}
```



stack canary hopes

overwrite return address $\implies$ overwrite canary

canary is secret

information disclosure (1a)

```
string command;
void vulnerable() {
    int value;
    for (;;) {
        cin >> command;
        if (command == "set") {
            cin >> value;
        } else if (command == "get") {
            cout << value << endl;
        } else if ...
    }
}
```

"get" command: can read *uninitialized value*

example: when I compiled this, `value` was stored on the stack

information disclosure (1b)

```
void vulnerable() {
    int value;
    ...
        } else if (command == "get") {
            cout << value << endl;
        }
    ...
}

void leak() {
    int secrets[] = {
        12345678, 23456789, 34567890,
        45678901, 56789012, 67890123,
    };
    cout << (void*) secrets << endl;
```

running this program (input in bold):

```
get
67890123
```

information disclosure (2)

```
void process() {
    char buffer[8] = "\0\0\0\0\0\0\0\0";
    char c = '␣';
    for (int i = 0; c != '\n' && i < 8; ++i) {
        c = getchar();
        buffer[i] = c;
    }
    printf("You␣input␣%s\n", buffer);
}
```

input `aaaaaaaa`

output `You input aaaaaaaa`*(whatever was on stack)*

information disclosure (3)

```
struct foo {
    char buffer[8];
    long *numbers;
};

void process(struct foo* thing) {
    ...
    scanf("%s", thing->buffer);
    ...
    printf("first␣number:␣%ld\n", thing->numbers[0]);
}
```

input: `aaaaaaaa`*(address of canary)*

address on stack *or* where canary is read from in thread-local storage

recall: ASLR

easlier mentioned ASLR (address space layout randomization)

for stack: choose secret starting address for stack

info disclosure bugs are a big problem for this!

exercise

```
struct point {
    int x, y, z;;
};
```

```
struct point *p;
...
    if (command == "get") {
        /* 'p' could be uninitialized */
        printf("%d,%d,%d\n", p->x, p->y, p->z);
    } ...
...
```

Which initial value for p (“left over” from prior use of register, etc.) would be most useful for figuring out the address of the stack pointer?

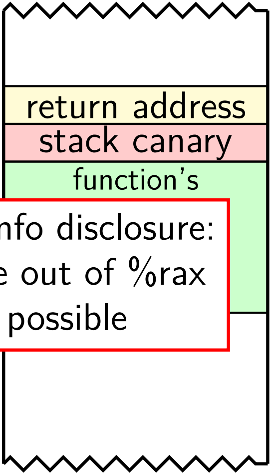
A. p is an invalid pointer and accessing it will crash the program

B. p points to space on the stack that is currently unallocated, but last contained an input buffer

C. p points to a struct allocated on the heap

compiler generated code

```
    pushq %rbx
    sub $0x20,%rsp
/* copy value from thread-local storage */
    mov $0x28,%ebx
    mov %fs:(%rbx),%rax
/* onto the stack */
    mov %rax,0x18(%rsp)
/* clear register holding value */
    xor %eax, %eax
    ...
    ...
/* copy value back from stack */
    mov 0x18(%rsp),%rax
/* xor to compare */
    xor %fs:(%rbx),%rax
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    ret
call_stack_chk_fail:
```



trying to avoid info disclosure:
get canary value out of %rax
as soon as possible

intuition: shadow stacks

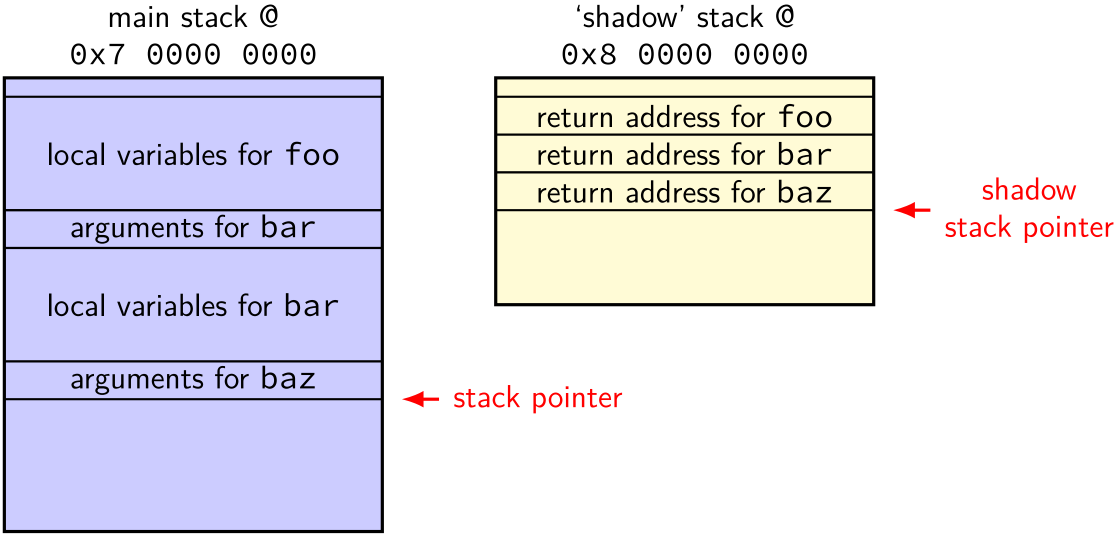
problem with stack: easy to leak address/values because used for lots of data

goal: keep sensitive data in **separate region**

easier to kepe address secret?

can use this for (stronger?) alternative to stack canaries

shadow stacks



implementing shadow stacks

bigger changes to compiler than canaries

more overhead to call/return from function

most commonly: store return address twice

shadow stacks on x86-64 (1)

idea 1: dedicate %r15 as shadow stack pointer,
copy RA to shadow stack pointer in function prologue

```
function:
    movq (%rsp), %rax       // RAX <- return address
    addq $-8, %r15          // R15 <- R15 - 8
    movq %rax, (%r15)       // M[R15] <- RAX
    ...
    movq (%rsp), %rdx        // RDX <- return address
    cmpq %rdx, (%r15)
    jne CRASH_THE_PROGRAM // if RDX != M[R15] goto
    add $8, %r15             // R15 <- R15 - 8
    ret
```

shadow stacks on x86-64 (2)

idea 2: dedicate %r15 as shadow stack pointer,
avoid normal call/return instruction

```
    addq $-8, %r15
    leaq after_call(%rip), %rax
    movq %rax, (%r15)
    jmp function
after_call:

function:
    ...
    addq $8, %r15           // R15 <- R15 + 8
    jmp *-8(%r15)           // jmp M[R15-8]
```

Android/AArch64 shadow stacks (1)

via `https://clang.llvm.org/docs/ShadowCallStack.html` (see also `https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html`)

dedicate register x18 to shadow stack pointer

x30 = return address (after ARM's call instruction (bl))

ARM call instruction saves return address in register...

without	with shadow stack

<pre>stp   x29, x30, [sp, #-16]!
mov   x29, sp
bl   bar
add   w0, w0, #1
ldp   x29, x30, [sp], #16
ret</pre>
	<pre>str x30, [x18], #8
	stp x29, x30, [sp, #-16]!
	mov x29, sp
	bl bar
	add w0, w0, #1
	ldp x29, x30, [sp], #16
	ldr x30, [x18, #-8]!
	ret</pre>

Intel CET shadow stacks

future Intel processor extension adds shadow stacks

- “Control-flow Enforcement Technology”

new shadow stack pointer

CALL/RET: push/pop from BOTH stacks

shadow stack pages are marked as read-only in page table

- cannot be written through normal instructions
- extra bit identifying as shadow stack (not “normal” read-only page)

preventing shadow stack writes?

ARM64 scheme: prevent writes if

- shadow stack pointer is never leaked (dedicated register)
- shadow stack random location can't be guessed (or queried otherwise)

Intel CET: prevent writes unless

- OS (priviliged/kernel mode) instructions to setup shadow stack used

can we prevent writes without relying on avoiding info leaks...
and without special hardware support?

- well, yes, but ...

some early stack canary benchmarks

from Chiueh and Hsu, “RAD: A Compile-Time Solution to Buffer Overflow Attacks” (2001)

Program size	Program tested	User time	System time	Real tim
11991 lines	Original ctags	0.57	0.05	0.62
	MineZone RAD-protected ctags	0.58	0.05	0.63
	Read-Only RAD-protected ctags	8.16	19.17	27.3

Table 3 Macro-benchmark results of ctags

Program size	Program tested	User time	System time	Real tim
4500 lines	Original gcc	3.53	0.19	3.72
	Mine Zone RAD-protected gcc	4.67	0.2	4.87
	Read-Only RAD-protected gcc	20.46	50.43	70.8

Table 4 Macro-benchmark results of gcc

automatic shadow stacks?

if we change how CALL/RET works…

…maybe we can add shadow stack support to existing programs?

- either with hardware support, or
- in software with emulation techniques?

well, there's a problem…

the problem in C++

```
void Foo() {
    try {
        ... Bar() ...
    } except (std::runtime_error &error) {
        ...
    }
}

void Bar() {
    ... Quux() ...
}
void Quux() {
    ...
    throw std::runtime_error("...");
    ...
}
```

the problem in C

```
jmp_buf env;
const char *error;
void Foo() {
    if (0 == setjmp(env)) {
        Bar();
    } else {
        ...
    }
}

void Bar() {
    ... Quux() ...
}
void Quux() {
    ...
    error = "...";
    longjmp(env, 1);
    ...
```

dealing with non-local returns

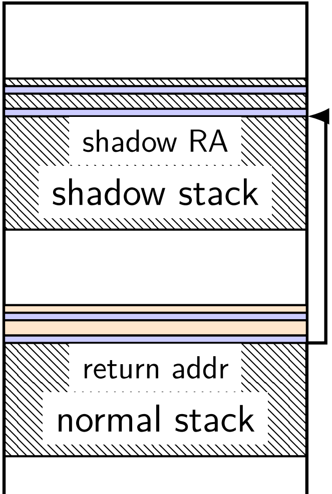
exceptions and setjmp/longjmp deliberately skip return calls

one solution: "direct" shadow stack

fixed (possibly secret) offset from normal stack

shadow stack only stores return addreses

space in between return addresses left as nulls



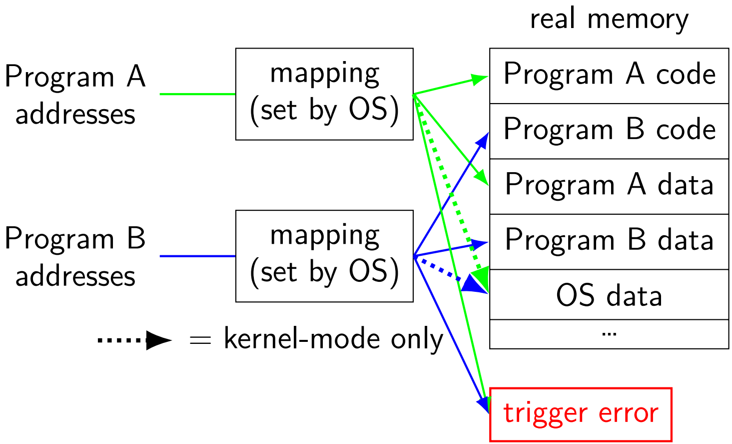
what do shadow stacks stop?

combined with a information leak that can dump arbitrary bytes of memory,
which of these exploits would shadow stacks stop...

A. using format string exploit to point stack return address to the 'system' function
B. using format string exploit to point VTable to the 'system' function
C. using an unchecked string copy that goes over the end of a stack buffer into the return address and pointing the return address to the 'system' function
D. using a buffer overflow that overwrites a saved stack pointer value to cause return to use a different address
E. using pointer subterfuge to overwrite the GOT entry for 'printf' to point to the 'system' function

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

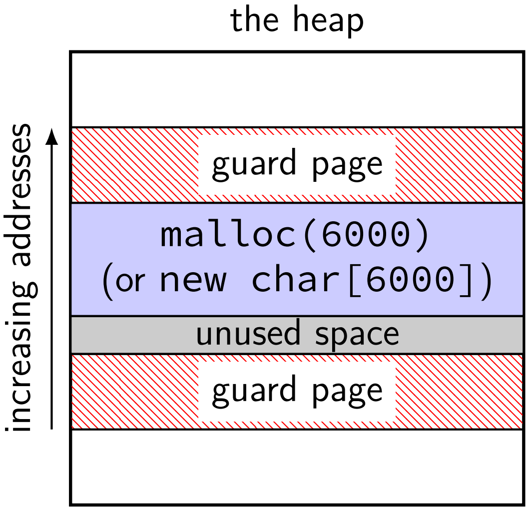
via *table*: OS decides *what memory program sees*

- 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

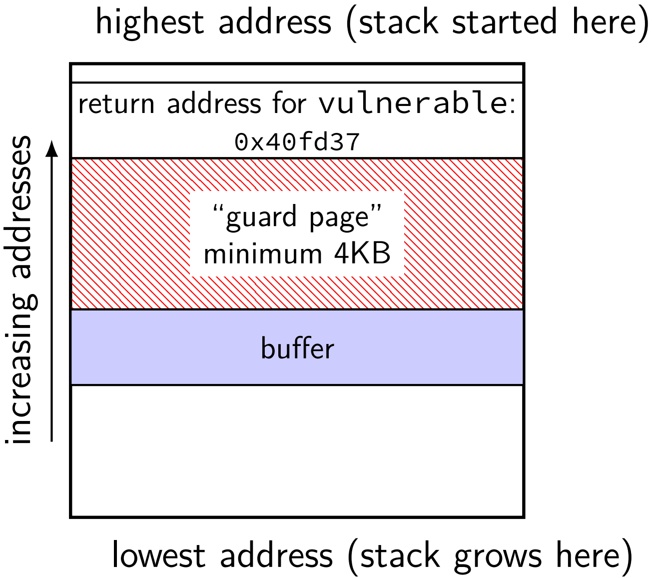
- commonly done by real malloc/new's for *large allocations*

problem: minimum actual allocation 4KB

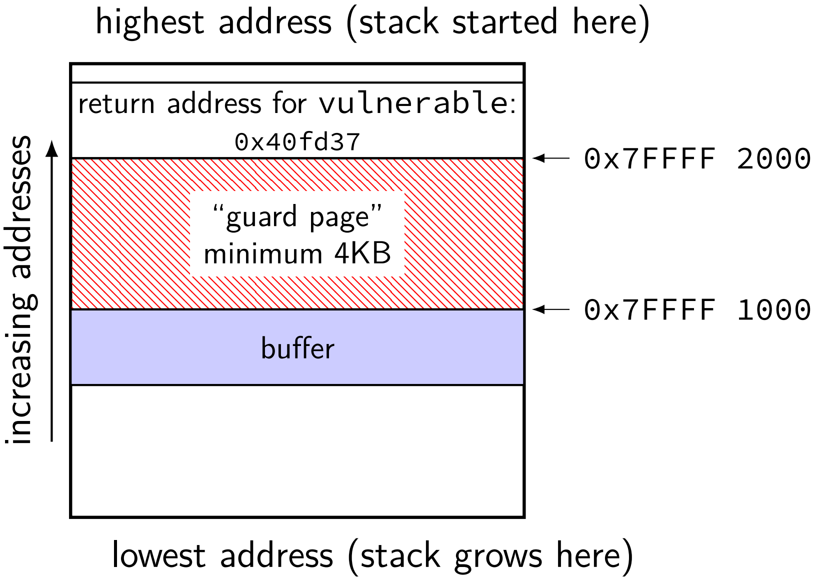
problem: substantially slower

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

stack canary alternative

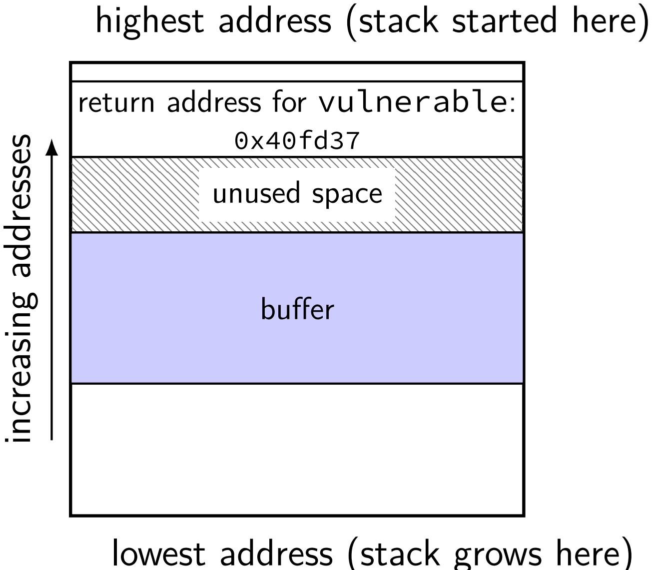


stack canary alternative

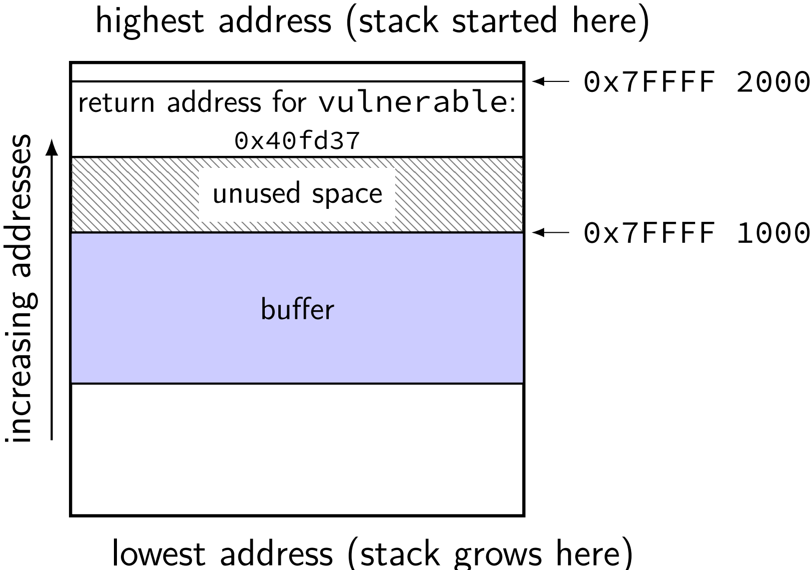


address	read	writ
0x7FFFF2000–0x7FFFF2FFF	yes	yes
0x7FFFF1000–0x7FFFF1FFF	no	no
0x7FFFF0000–0x7FFFF0FFF	yes	yes

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

recall: relocation

```
.data
string: .asciz "Hello,␣World!"
.text
main:
    movq $string, %rdi /* NOT PC/RIP-relative mov */
```

generates: `(objdump --disassemble --reloc)`

```
   0:    48 c7 c7 00 00 00 00       mov     $0x0,%rdi
                        3: R_X86_64_32S .data
```

relocation record says how to fix `0x0` in `mov`

- `3`: location in machine code
- `R_X86_64_32S`: 32-bit signed integer
- `.data`: address to insert