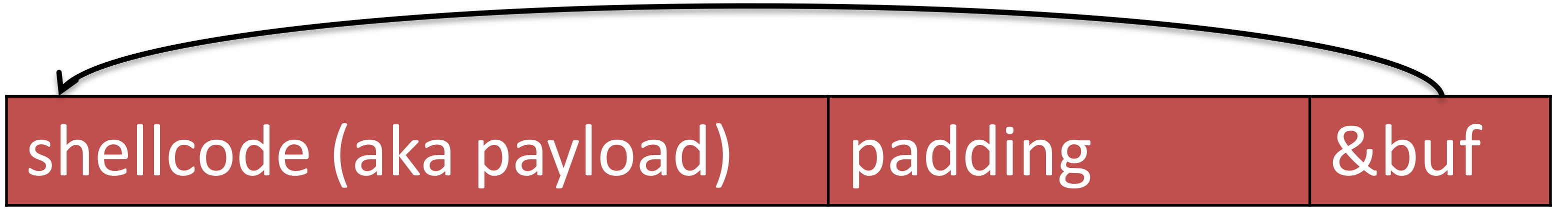
Lecture 11 – Control-flow Hijacking Defenses

Stephen Checkoway
Oberlin College
Slides adapted from Miller, Bailey, and Brumley

Control Flow Hijack:
Always control + computation



computation + ***control***

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- …

Same principle, different mechanism

Control Flow Hijacks

… happen when an attacker gains control of

***the instruction pointer**.*

Two common hijack methods:

- buffer overflows
- format string attacks

Control Flow Hijack Defenses

Bugs are the root cause of hijacks!

- Find bugs with analysis tools
- Prove program correctness

Mitigation Techniques:

- Canaries
- Data Execution Prevention/No eXecute
- Address Space Layout Randomization

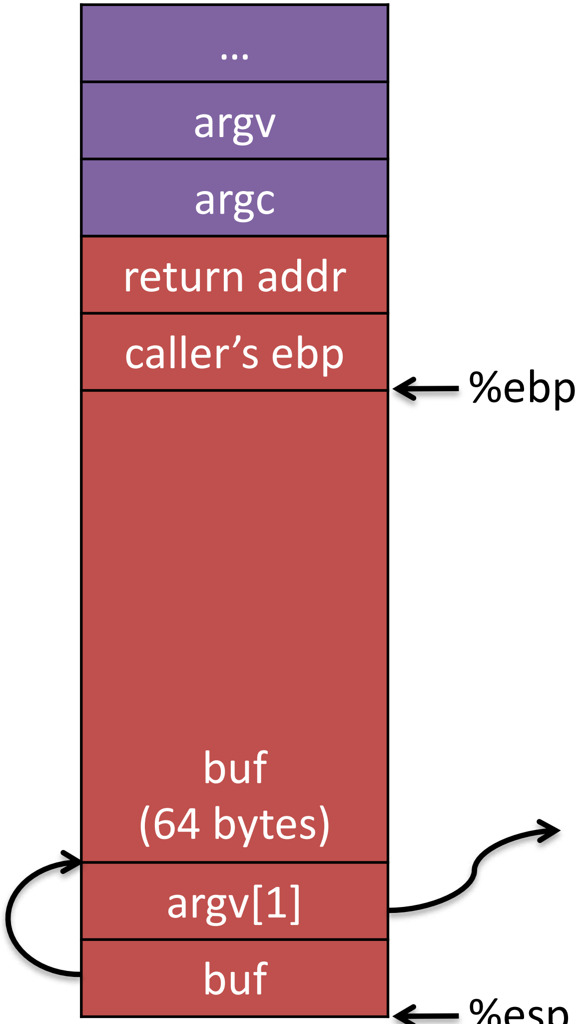
CANARY / STACK COOKIES

http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

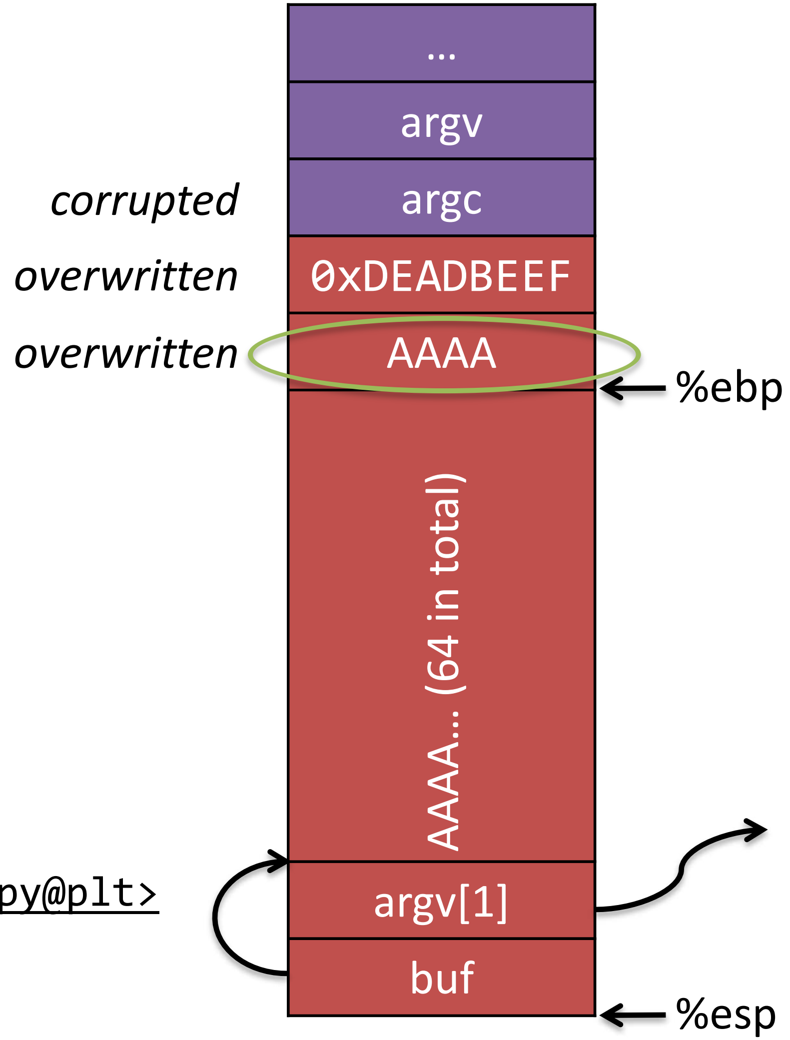


6

“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



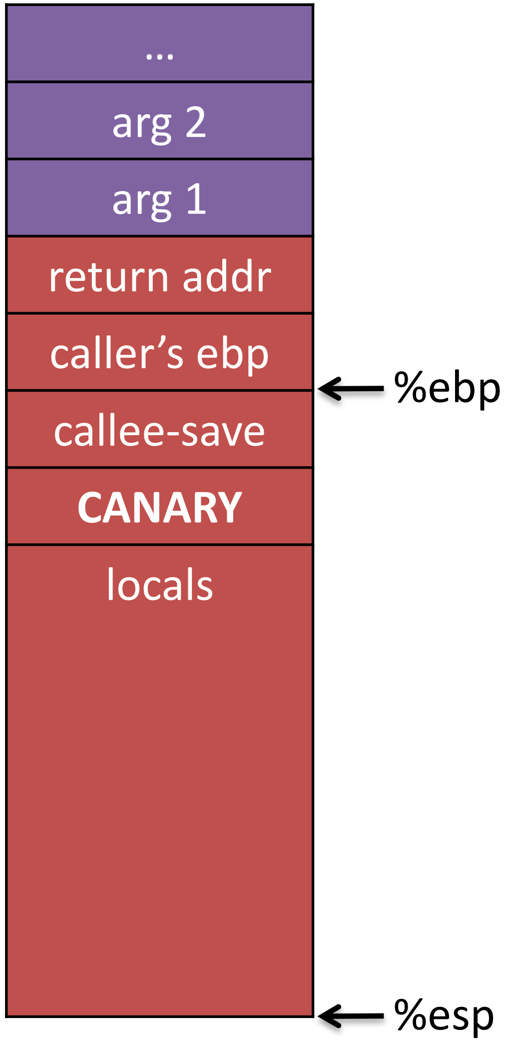
7

StackGuard [Cowen etal. 1998]

Idea:

- prologue introduces a ***canary word*** between return addr and locals
- epilogue checks canary before function returns

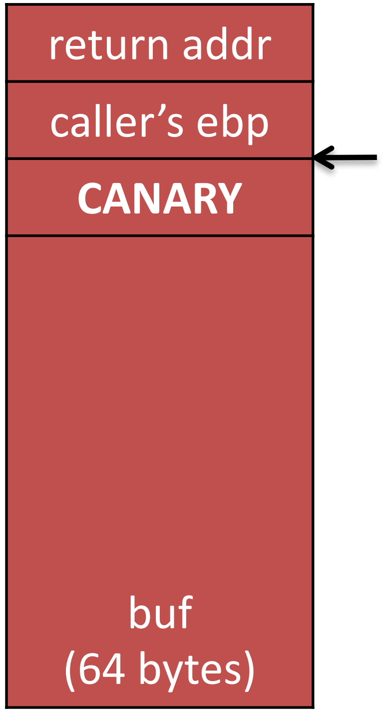
Wrong Canary => Overflow



gcc Stack-Smashing Protector (ProPolice)

```
Dump of assembler code for function main:
   0x08048440 <+0>:  push   %ebp
   0x08048441 <+1>:  mov    %esp,%ebp
   0x08048443 <+3>:  sub    $76,%esp
   0x08048446 <+6>:  mov    %gs:20,%eax
   0x0804844c <+12>: mov    %eax,-4(%ebp)
   0x0804844f <+15>: xor    %eax,%eax
   0x08048451 <+17>: mov    12(%ebp),%eax
   0x08048454 <+20>: mov    4(%eax),%eax
   0x08048457 <+23>: mov    %eax,4(%esp)
   0x0804845b <+27>: lea    -68(%ebp),%eax
   0x0804845e <+30>: mov    %eax,(%esp)
   0x08048461 <+33>: call   0x8048350 <strcpy@plt>
   0x08048466 <+38>: mov    -4(%ebp),%edx
   0x08048469 <+41>: xor    %gs:20,%edx
   0x08048470 <+48>: je     0x8048477 <main+55>
   0x08048472 <+50>: call   0x8048340 <__stack_chk_fail@plt>
   0x08048477 <+55>: leave
   0x08048478 <+56>: ret
```

Compiled with v4.6.1:
`gcc -fstack-protector -O1 …`



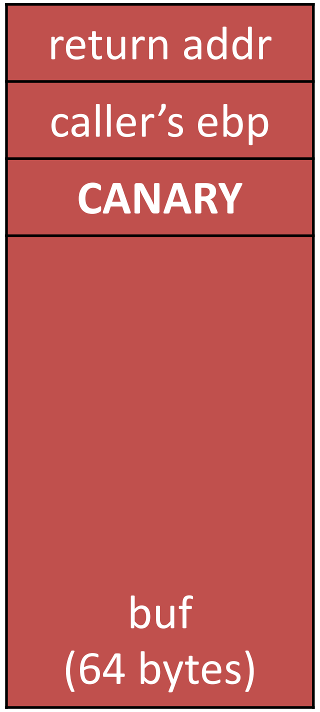
Canary should be **HARD** to Forge

- Terminator Canary
  - 4 bytes: 0,CR,LF,-1 (low->high)
  - terminate `strcpy()`, `gets()`, …

- Random Canary
  - 4 random bytes chosen at load time
  - stored in a guarded page
  - need good randomness

Ideas for defeating stack canaries?

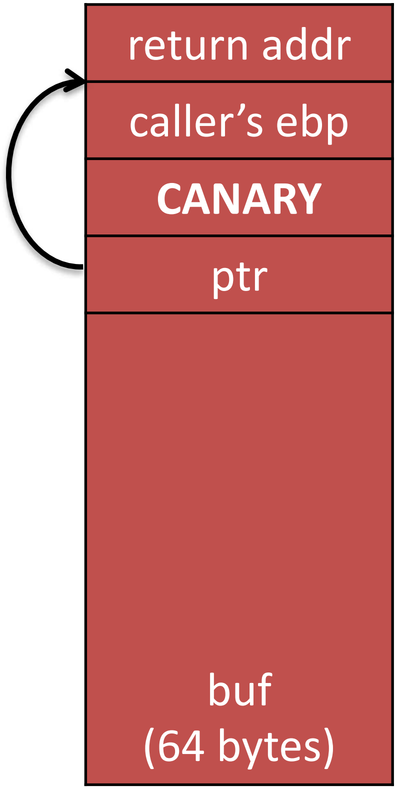
- Use targeted write, e.g., format string
- Overwrite data pointer first
- Overwrite function pointer loaded and used from higher up the stack
- memcpy buffer overflow with fixed canary
- Canary leak



Bypass: Data Pointer Subterfuge

Overwrite a data pointer *first*…

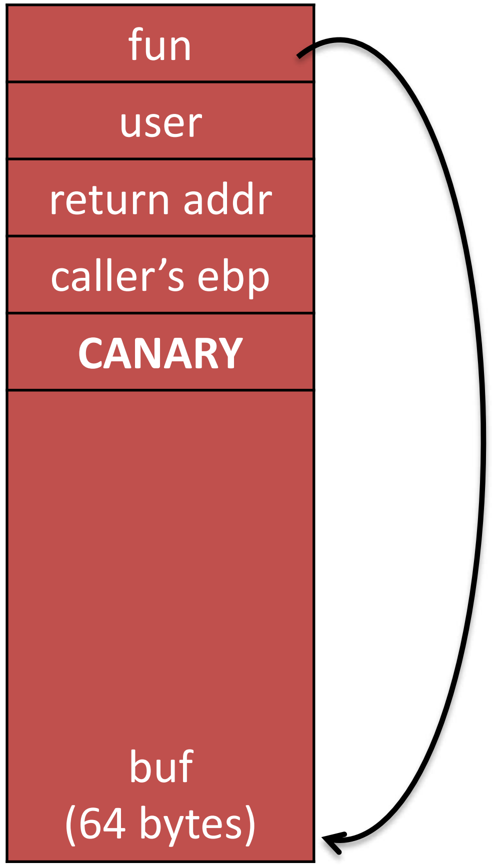
```
int *ptr;
char buf[64];
memcpy(buf, user1);
*ptr = user2;
```



Overwrite function pointer higher up

```
void contrived(const char *user, void (*fun)(char *)) {
    char buf[64];
    strcpy(buf, user);
    fun(buf);
}
```

- Overflow buffer to overwrite fun on the stack
- Tricky! Compiler can load fun into a register before strcpy (this can happen with optimization)
- Works better with structs with function pointers (e.g., OpenSSL) or C++ classes



memcpy/memmove with fixed canary

- Fixed canary values like 00 0d 0a ff (0, CR, NL, -1) are designed to terminate string operations like strcpy and gets
- However, they are trivial to bypass with memcpy vulnerabilities

Canary leak I: two vulnerabilities

- Exploit one vulnerability to read the value of the canary
- Exploit a second to perform a buffer overflow on the stack, overwriting the canary with the correct value

Canary leak II: pre-fork servers

- Some servers fork worker processes to handle connections
- In the main server process
  - Establish listening socket
  - Fork all the workers; if any die, fork a new one
- In the worker process (in a loop)
  - Accept a connection on the listening socket
  - Process request

Canary leak II: pre-fork servers

- This design interacts poorly with stack canaries
- Since each worker is forked from the main process, it initially has exactly the same memory layout and contents, including stack canary values!
- Attacker can often learn the canary a byte at a time by overflowing just a single byte of the canary, trying values 00 through ff until it doesn't crash; then move on to the next byte

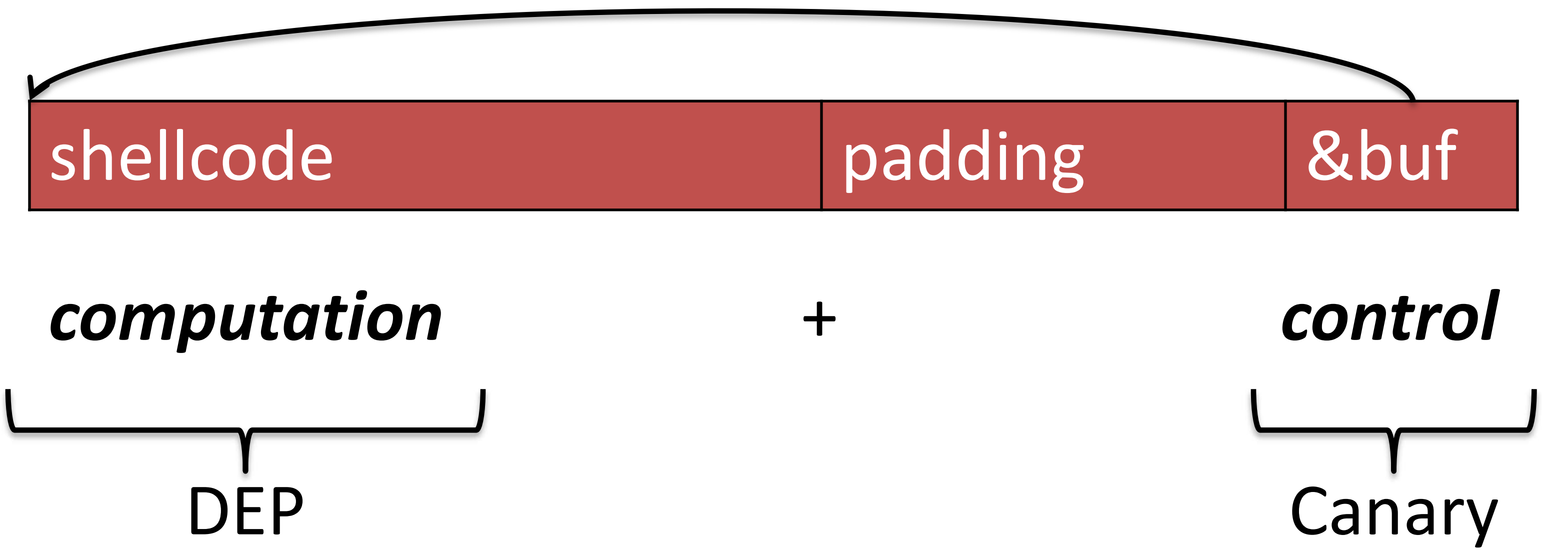
What is "Canary"?

Wikipedia: "the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system."

lecture

DATA EXECUTION PREVENTION (DEP) /
NO EXECUTE (NX)/
EXECUTE DISABLED (XD)/
EXECUTE NEVER (XN)

How to defeat exploits?



AMD, Intel put antivirus tech into chips

The companies plan to soon release technology that will allow processors to stop many computer attacks before they occur.

ByMichael Kanellos | January 8, 2004 -- 23:22 GMT (15:22 PST) | Topic: Intel

LAS VEGAS--Advanced Micro Devices and Intel plan to soon release technology that will allow processors to stop many attacks before they occur.

Execution Protection by AMD, technology contained in AMD's Athlon 64 chips, prevents a buffer overflow, a common method used to attack computers. A buffer overflow essentially overwhelms a computer's defense systems and then inserts a malicious program in memory that the processor subsequently executes.

With Execution Protection, data in the buffer can only be read and, therefore, is prevented from doing its dirty work, John Morris, director of marketing at AMD, said in an interview Thursday at the Consumer Electronics Show here.

RELATED S

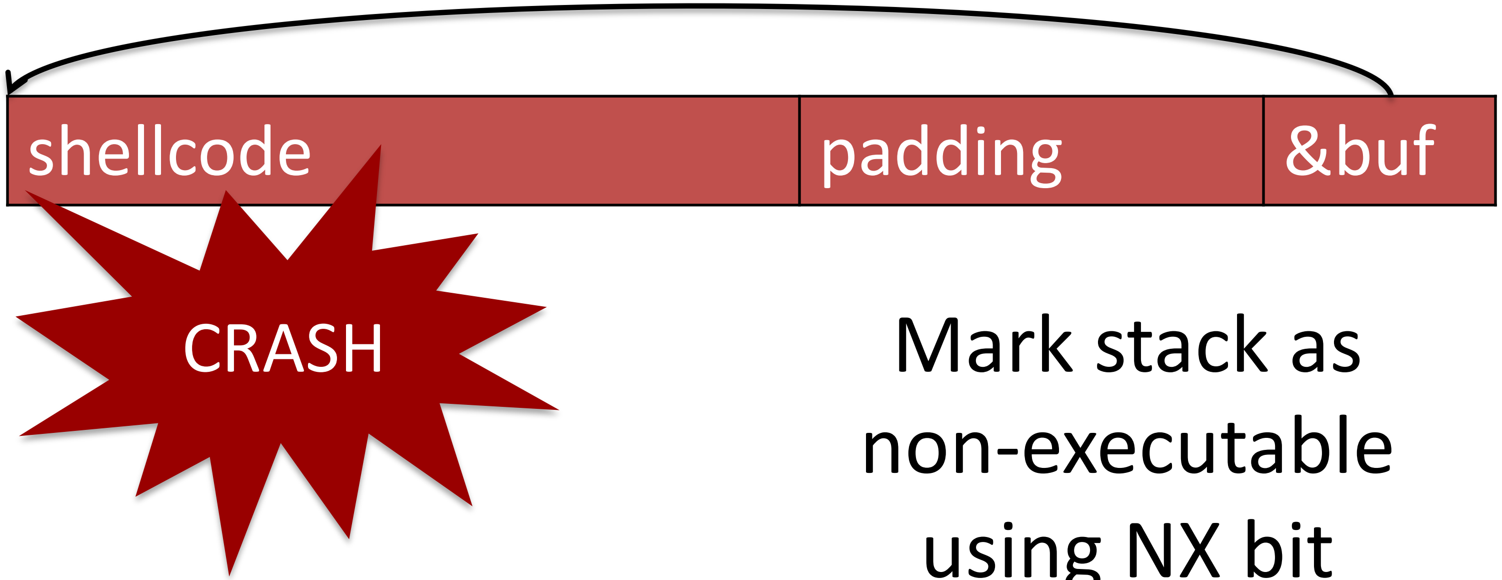
Internet
Intel lau
retail pla
million r
investm

Hardwar

Memory permissions

- Set (or clear) a bit in a page table entry to prevent code from being executed
- Enforced by hardware: Trying to fetch an instruction from a page marked as non-executable causes a processor fault

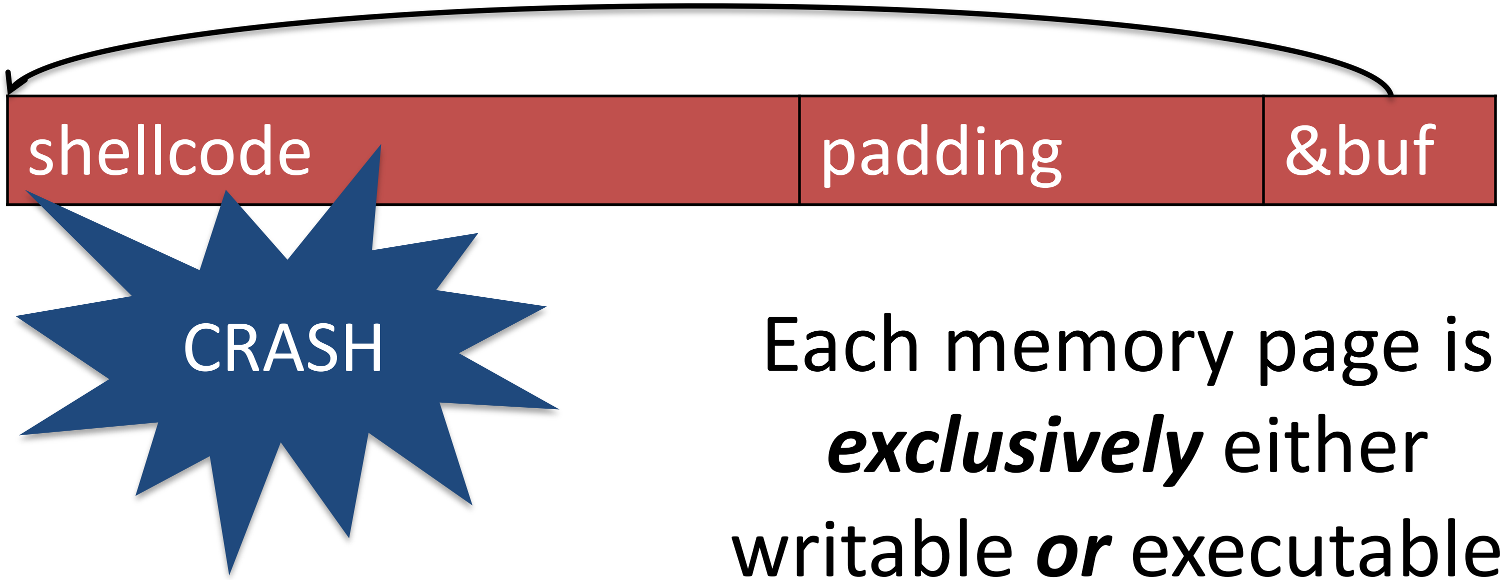
Data Execution Prevention



Mark stack as
non-executable
using NX bit

(still a Denial-of-Service attack!)

W ^ X



Each memory page is ***exclusively*** either writable ***or*** executable.

(still a Denial-of-Service attack!)

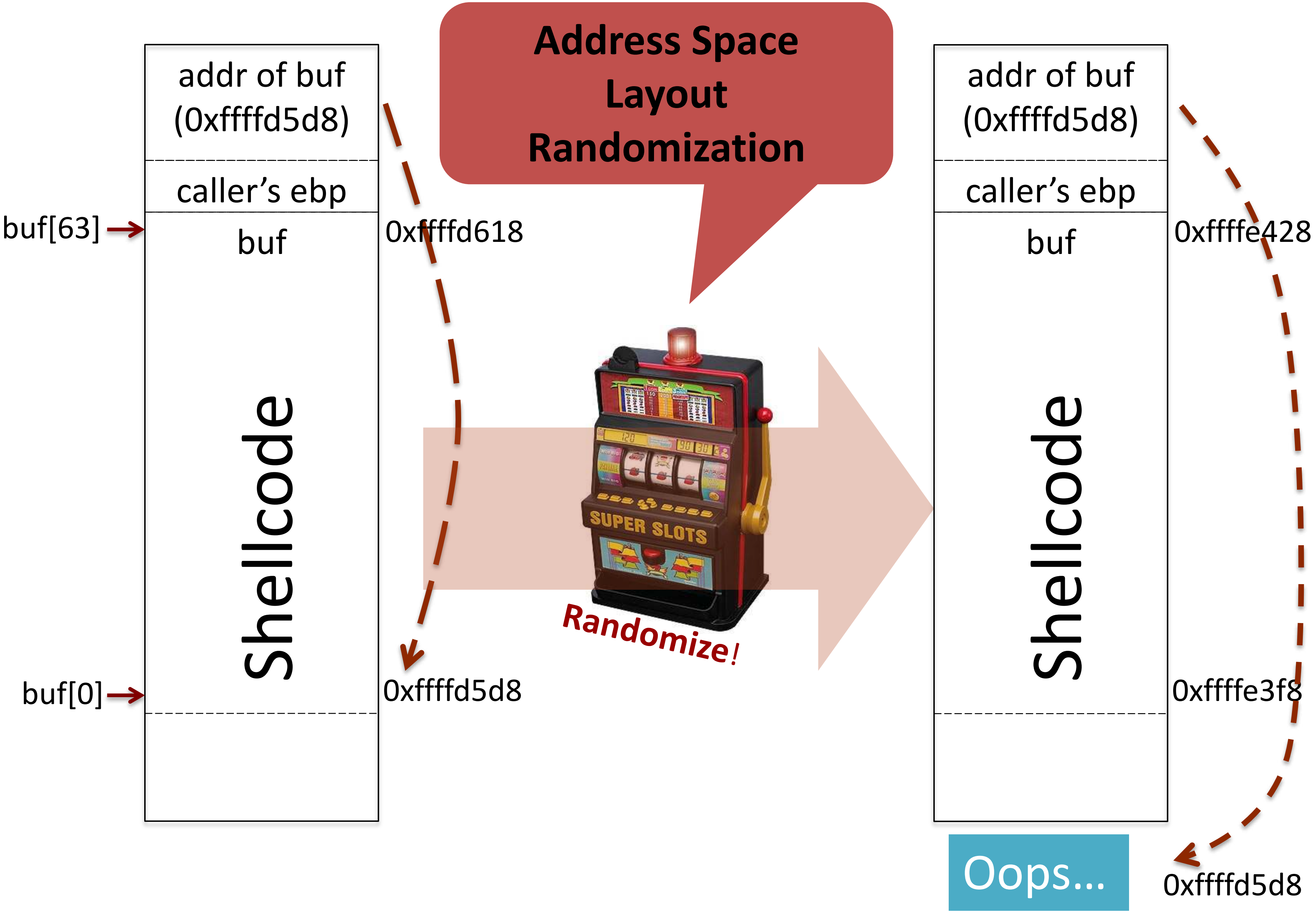
Actually a pretty old idea

- MIPS R2000 (from 1986) has per-page readable, writable, executable bits
- Intel 80386 (from 1985) does not. Mapped pages are always readable and executable
- Intel 80286 (from 1982) introduced 16-bit “protected mode” where code, data, and stack segments can be separated
- The 386 has a 32-bit “protected mode” but most OSes set code, data, and stack segments to be the entire virtual address space

Physical Address Extension

- Intel added an extension to increase the size of allowable physical memory beyond 4 GB
- PAE changed the page table format, added a third level of translation, and added the execute disable bit (but the OS has to enable both PAE and NX support)
- x86-64 uses the PAE format and thus supports NX

ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)



Address Space Layout Randomization
addr of buf (0xffffd5d8)
caller's ebp
buf[63]
buf
0xffffd618
Shellcode
buf[0]
0xffffd5d8
SUPER SLOTS
Randomize!
addr of buf (0xffffd5d8)
caller's ebp
buf
0xffffe428
Shellcode
0xffffe3f8
Oops...
0xffffd5d8

ASLR

Traditional exploits need precise addresses

- *stack-based overflows:* location of shell code
- *return-to-libc:* library addresses (we'll talk about this next time)

- **Problem:** program's memory layout is fixed
  - stack, heap, libraries etc.

- **Solution:** randomize addresses of each region!

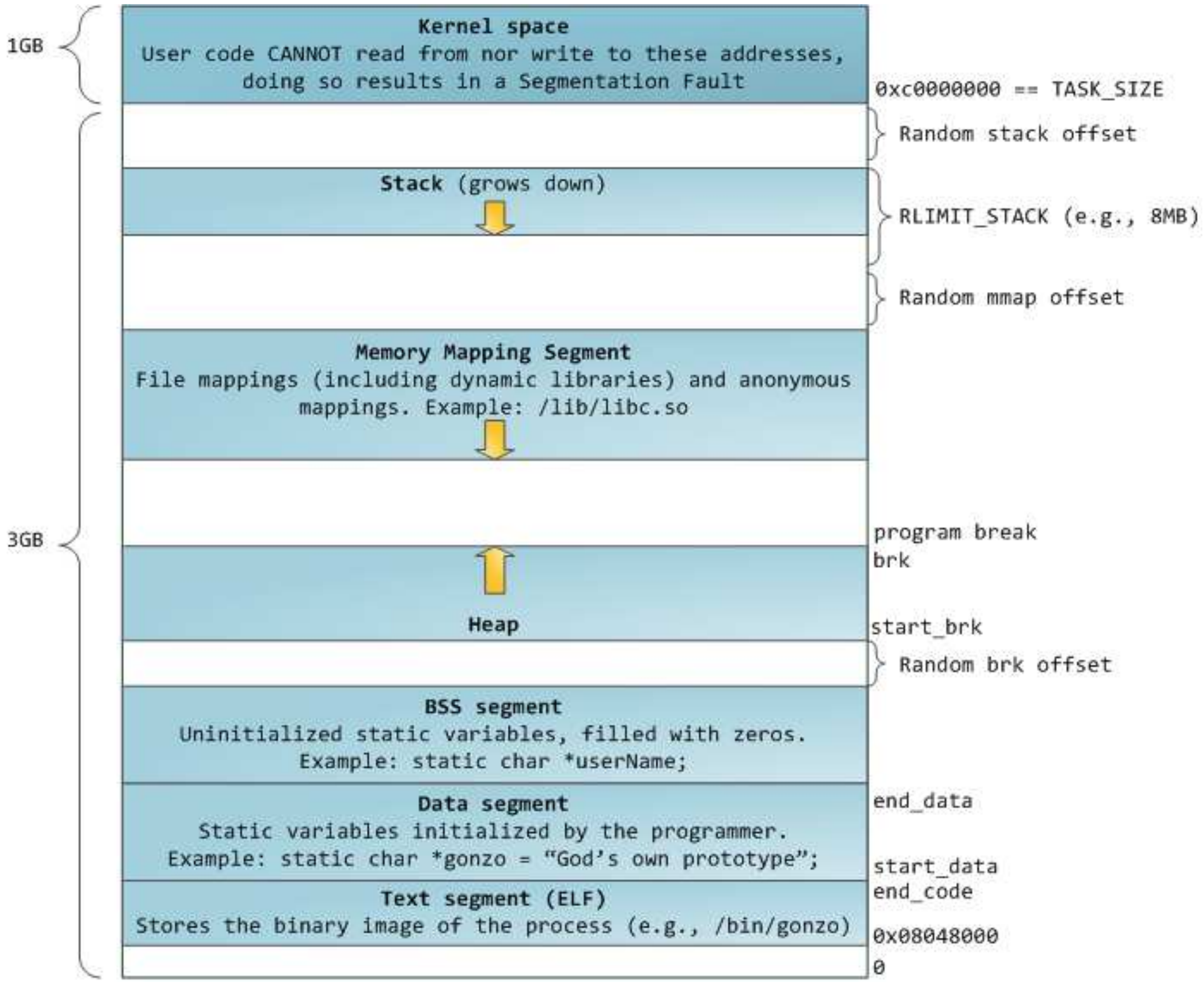


Image source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Running `cat` Twice

- Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

- Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

Bits of randomness (32-bit x86)

- Depends on the OS, but roughly
  - Program code and data: 0 bits (fixed addresses)
  - Heap: 13 bits (2^13 possible start locations)
  - Stack: 19 bits (2^19 possible start locations)
  - Libraries: 8 bits (2^8 possible start locations)
- With position-independent executables (PIE)
  - Program code and data: 8 bits
  - Others the same
- 64-bit has much more randomness

Support for ASLR added over time

- Initially by the PaX team for Linux
- All major OSes support it for applications
- Kernel ASLR now supported by major OSes

Is DEP + ASLR a panacea?

- Not really
- Next time: DEP bypass via code reuse attacks
- How can we bypass ASLR?

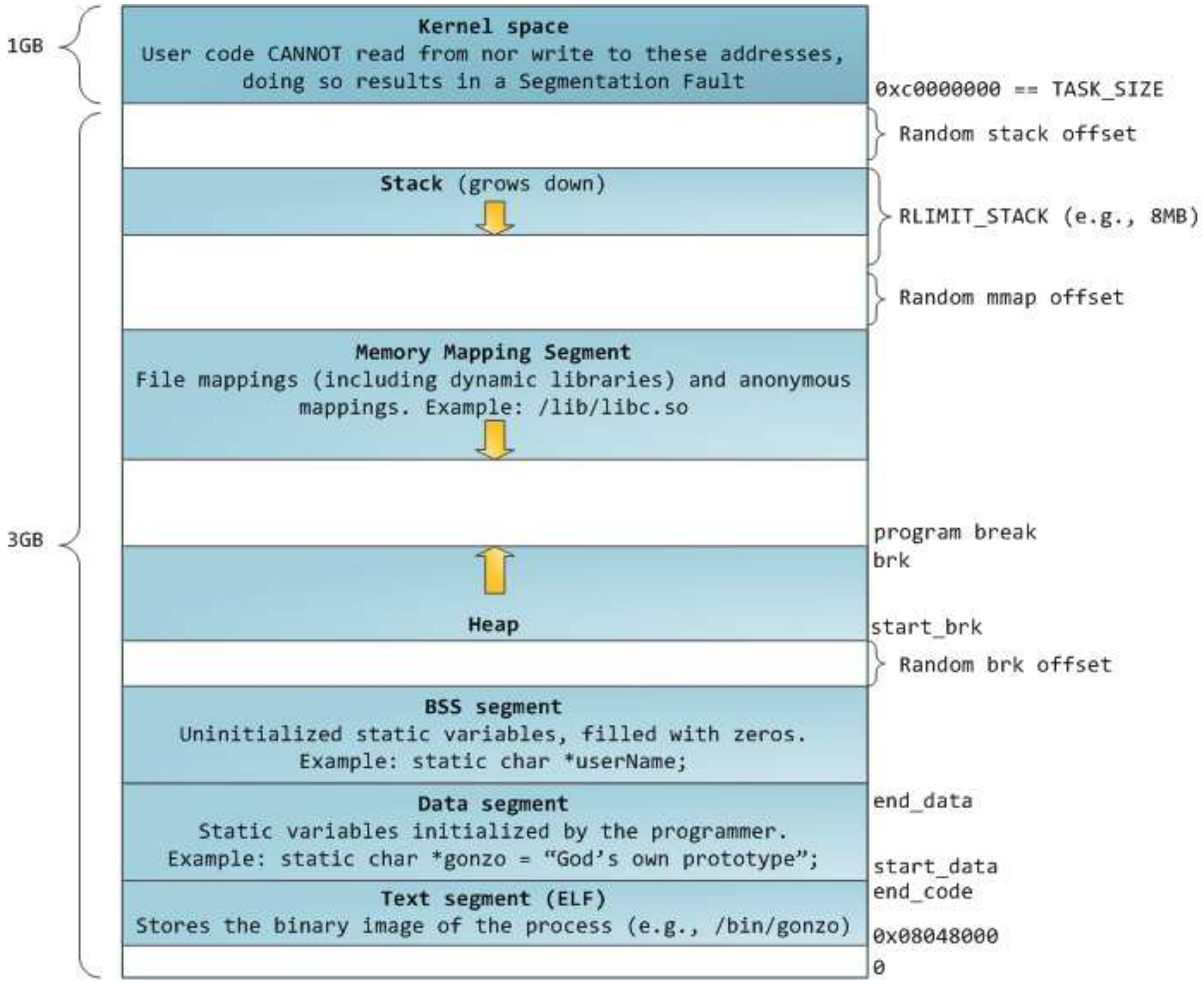


Image source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Bypassing ASLR

- Non-PIE binaries have fixed code and data addresses
- Each region has a random offset, but fixed layout => learning a single address in a region gives every address in the region
- Older Linux would let local attackers read the stack start address from /proc/<pid>/stat
- Servers that re-spawn (even with new randomization) can be brute forced when number of bits of randomness is low