Lecture 11 – Control-flow Hijacking Defenses

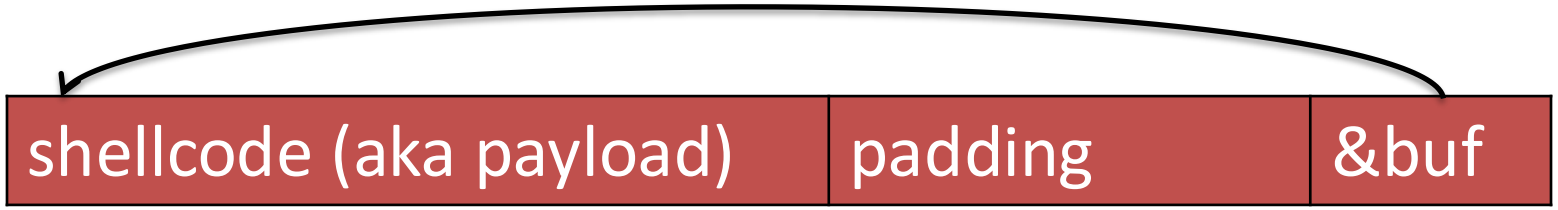
Stephen Checkoway
Oberlin College
Slides adapted from Miller, Bailey, and Brumley

Today

We’re going to look at 3 control-flow hijacking defenses

- Stack canaries/cookies
- Data execution prevention (DEP)/write XOR execute (w^x)
- Address space layout randomization (ASLR)

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

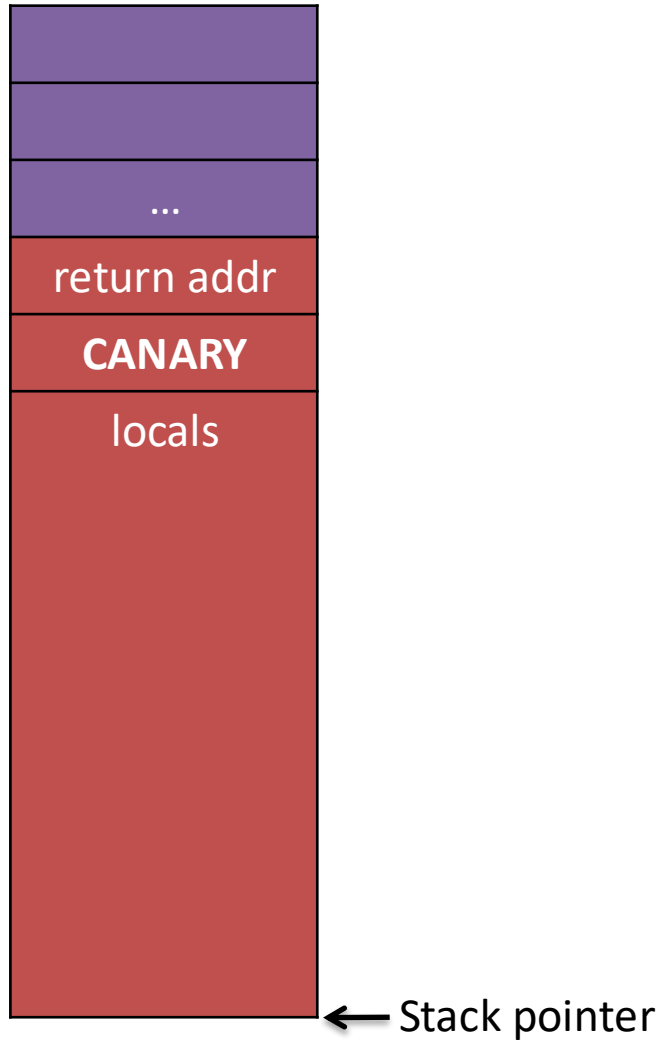
http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

StackGuard [Cowen etal. 1998]

Idea:

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

Wrong Canary => Overflow



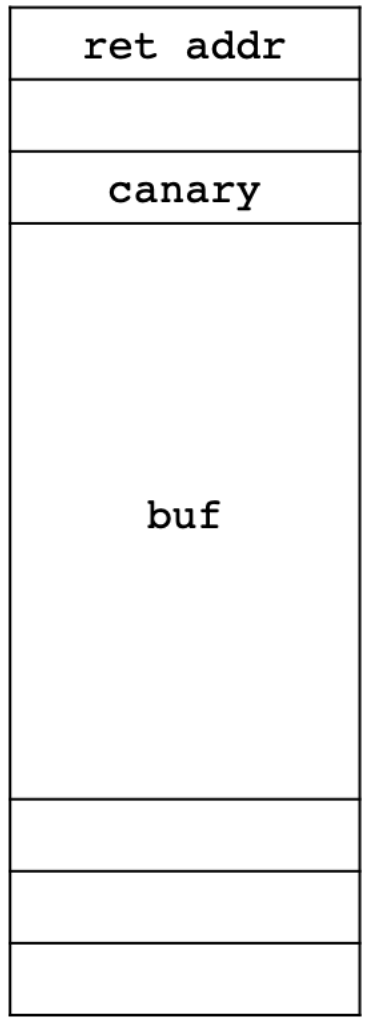
Stack Canaries

```
int bar(char *);
char foo(void) {
    char buf[100];
    bar(buf);
    return buf[0];
}
```

```
foo:
        sub     rsp, 120
        mov     rax, QWORD PTR fs:40
        mov     QWORD PTR [rsp+104], rax
        xor     eax, eax
        mov     rdi, rsp
        call    bar
        movzx   eax, BYTE PTR [rsp]
        mov     rdx, QWORD PTR [rsp+104]
        sub     rdx, QWORD PTR fs:40
        jne     .L5
        add     rsp, 120
        ret
.L5:
        call    __stack_chk_fail
```

fs is a segment register which, on x86-64 is essentially just a (complicated) pointer

fs:40 means take the value 40 bytes from the start of the fs segment



Aside: where the canary lives

- The canary is stored in the “dynamic thread vector”
- On x86-64, that’s accessible using the fs segment register

- fs:0 is a pointer to itself which can be used to get the actual “linear” address
- fs:40 is where the canary lives

Printing the canary

```
static void print_canary(int thread_num) {
    unsigned long canary = 0;
    unsigned long *dtv = 0;
    asm("movq %%fs:40,%0" : "=r"(canary));
    asm("movq %%fs:0,%0" : "=r"(dtv));
    assert(canary == dtv[5]);
    printf("Thread %d: canary=%016lx dtv=%p\n",
           thread_num, canary, dtv);
}
```

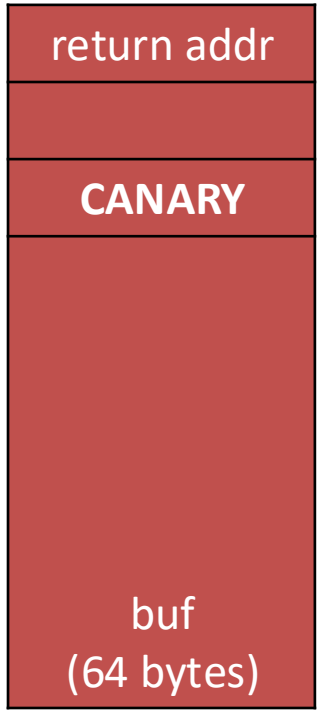
```
user@sandbox:/tmp$ ./a.out
Thread 0: canary=39aee18d91a43700 dtv=0x7f3ce9c4e740
Thread 3: canary=39aee18d91a43700 dtv=0x7f3ce8c4b6c0
Thread 2: canary=39aee18d91a43700 dtv=0x7f3ce944c6c0
Thread 1: canary=39aee18d91a43700 dtv=0x7f3ce9c4d6c0
user@sandbox:/tmp$ ./a.out
Thread 0: canary=0551b72919271000 dtv=0x7fc1593e3740
Thread 3: canary=0551b72919271000 dtv=0x7fc1583e06c0
Thread 2: canary=0551b72919271000 dtv=0x7fc158be16c0
Thread 1: canary=0551b72919271000 dtv=0x7fc1593e26c0
user@sandbox:/tmp$ ./a.out
Thread 0: canary=c61bc68a241ec900 dtv=0x7fb9fbd02740
Thread 3: canary=c61bc68a241ec900 dtv=0x7fb9facff6c0
Thread 2: canary=c61bc68a241ec900 dtv=0x7fb9fb5006c0
Thread 1: canary=c61bc68a241ec900 dtv=0x7fb9fbd016c0
```

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?

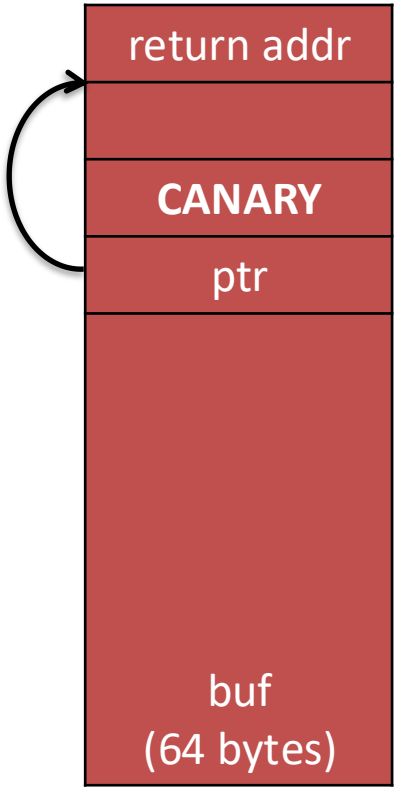
- Use targeted write, e.g., format string
- Overwrite data pointer first
- Overwrite function pointer loaded and used from higher up the stack
- memcpy buffer overflow with fixed canary
- Canary leak



Bypass: Data Pointer Subterfuge

Overwrite a data pointer *first*…

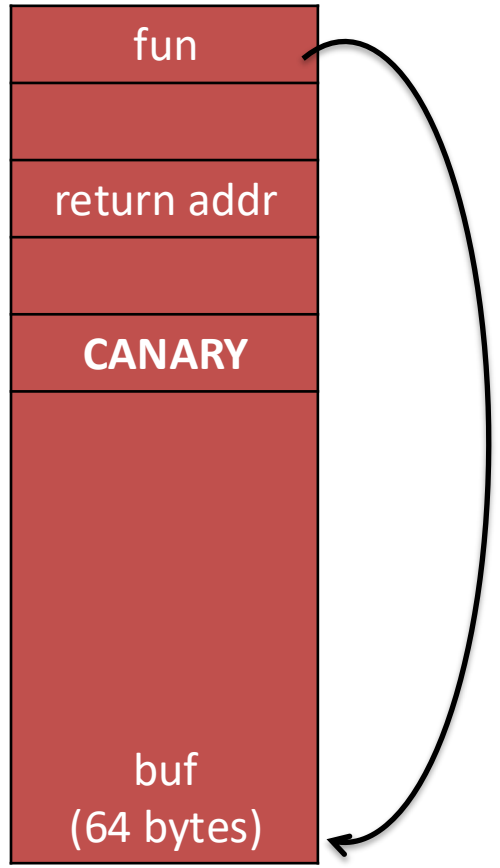
```
long *ptr;
char buf[64];
memcpy(buf, user1, 72);
*ptr = user2;
```



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

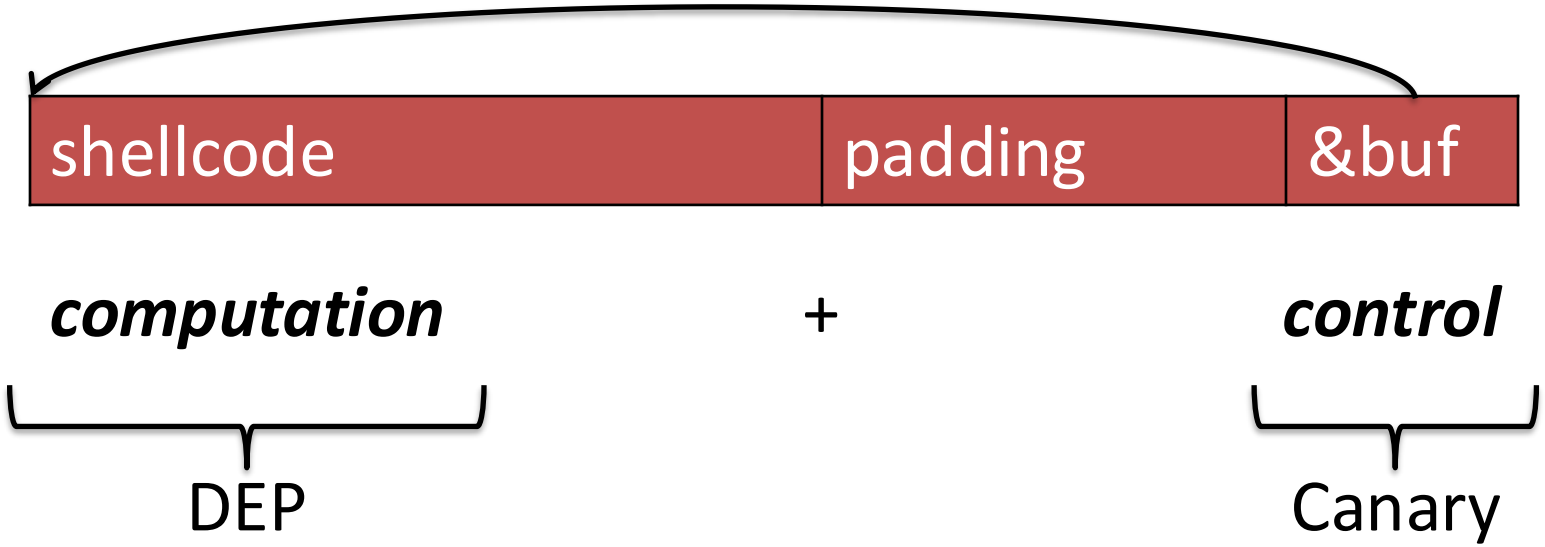
- Some servers fork worker processes to handle connections
- In the main server process
  - Establish listening socket
  - Fork all the workers; if any die, fork a new one
- In the worker process (in a loop)
  - Accept a connection on the listening socket
  - Process request

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

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.

By Michael 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.

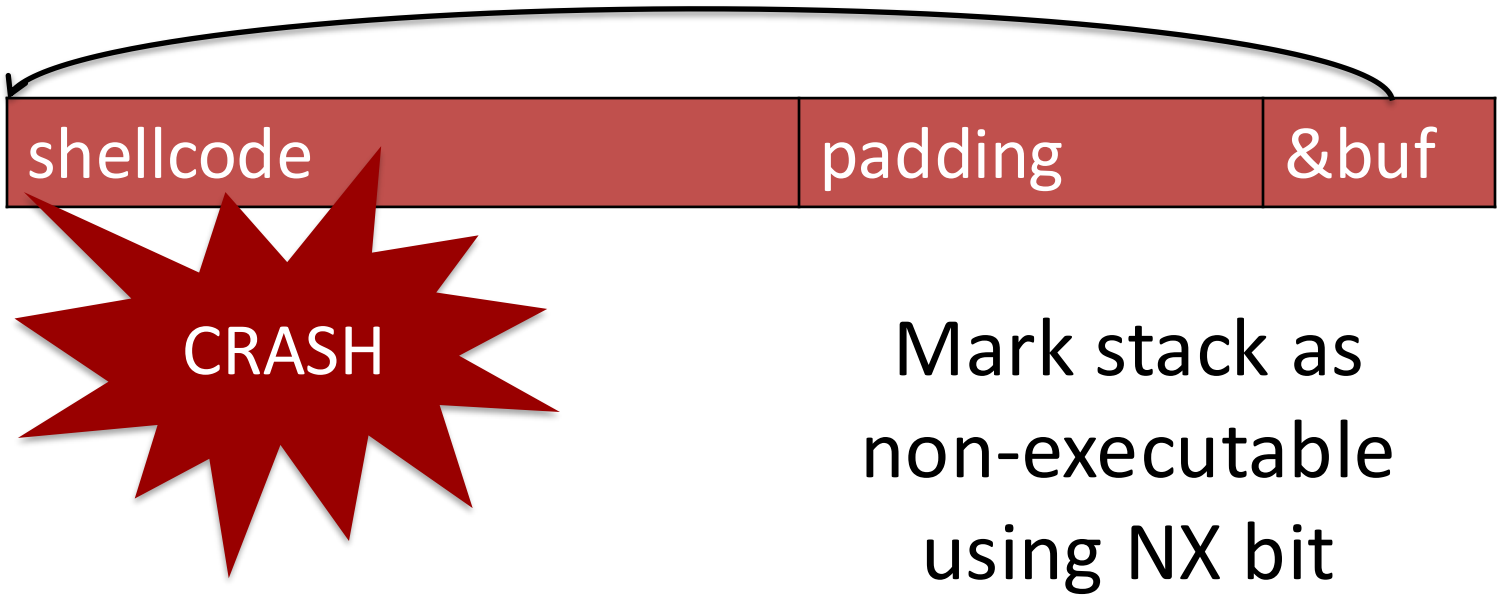
Execution Protection by AMD, technology contained in AMD's Athlon 64 chips, prevents a buffer overflow, a common method used to attack computers. A buffer overflow essentially overwhelms a computer's defense systems and then inserts a malicious program in memory that the processor subsequently executes.

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.

Memory permissions

- Set (or clear) a bit in a page table entry to prevent code from being executed
- This bit is called
  NX (No eXecute) by AMD,
  XD (eXecute Disabled) by Intel
  XN (eXecute Never) by ARM
  XI (eXecute Inhibit) by MIPS
  Most people just call it the NX bit
- Enforced by hardware: Trying to fetch an instruction from a page marked as non-executable causes a processor fault

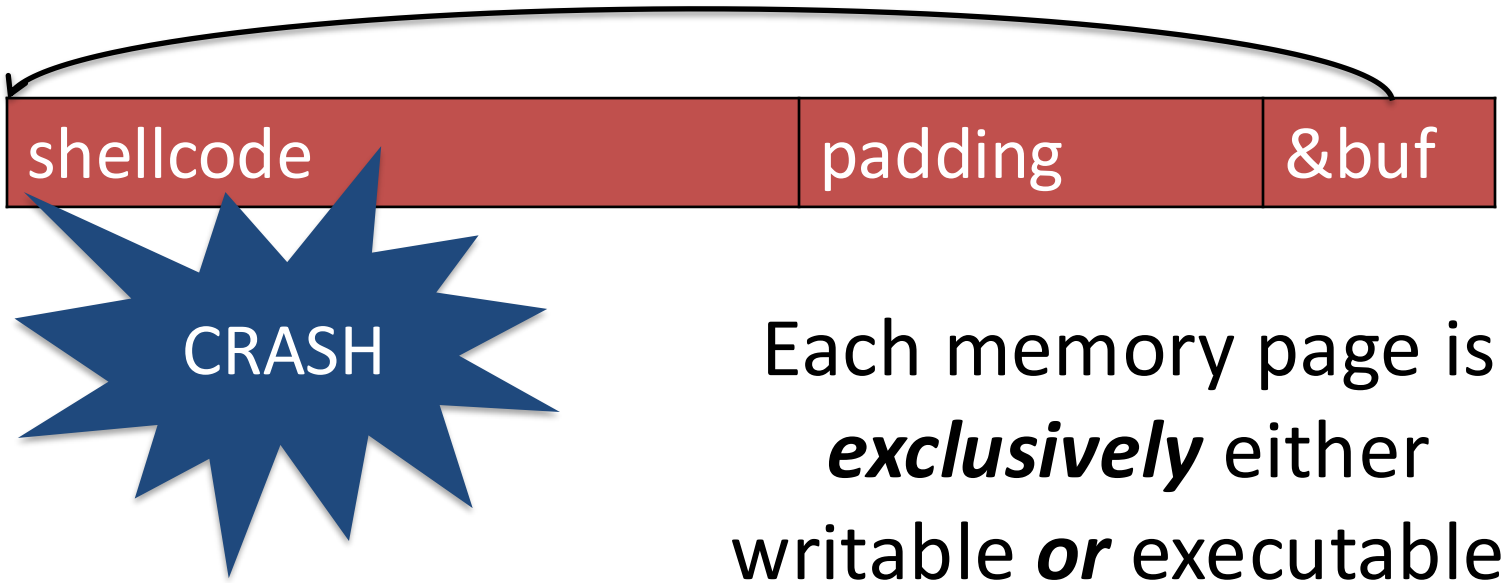
Data Execution Prevention



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!)

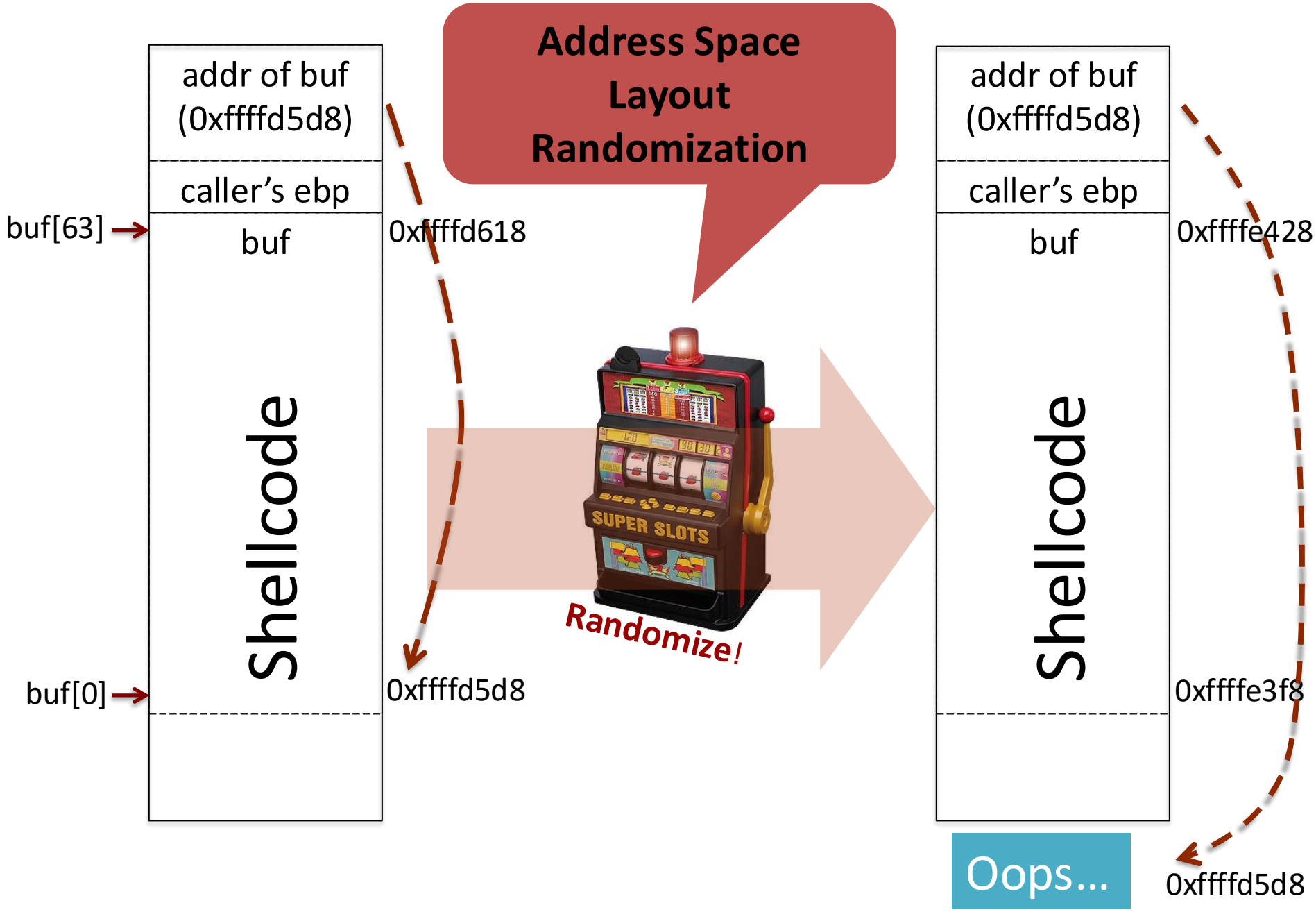
Actually a pretty old idea

- 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
- x86-64 *always* treats code, data, and stack segments to be the whole 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 (with 4 or 5 levels of page tables) and thus supports NX

ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)


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

ASLR demo

- $ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
- $ watch -d cat /proc/self/maps

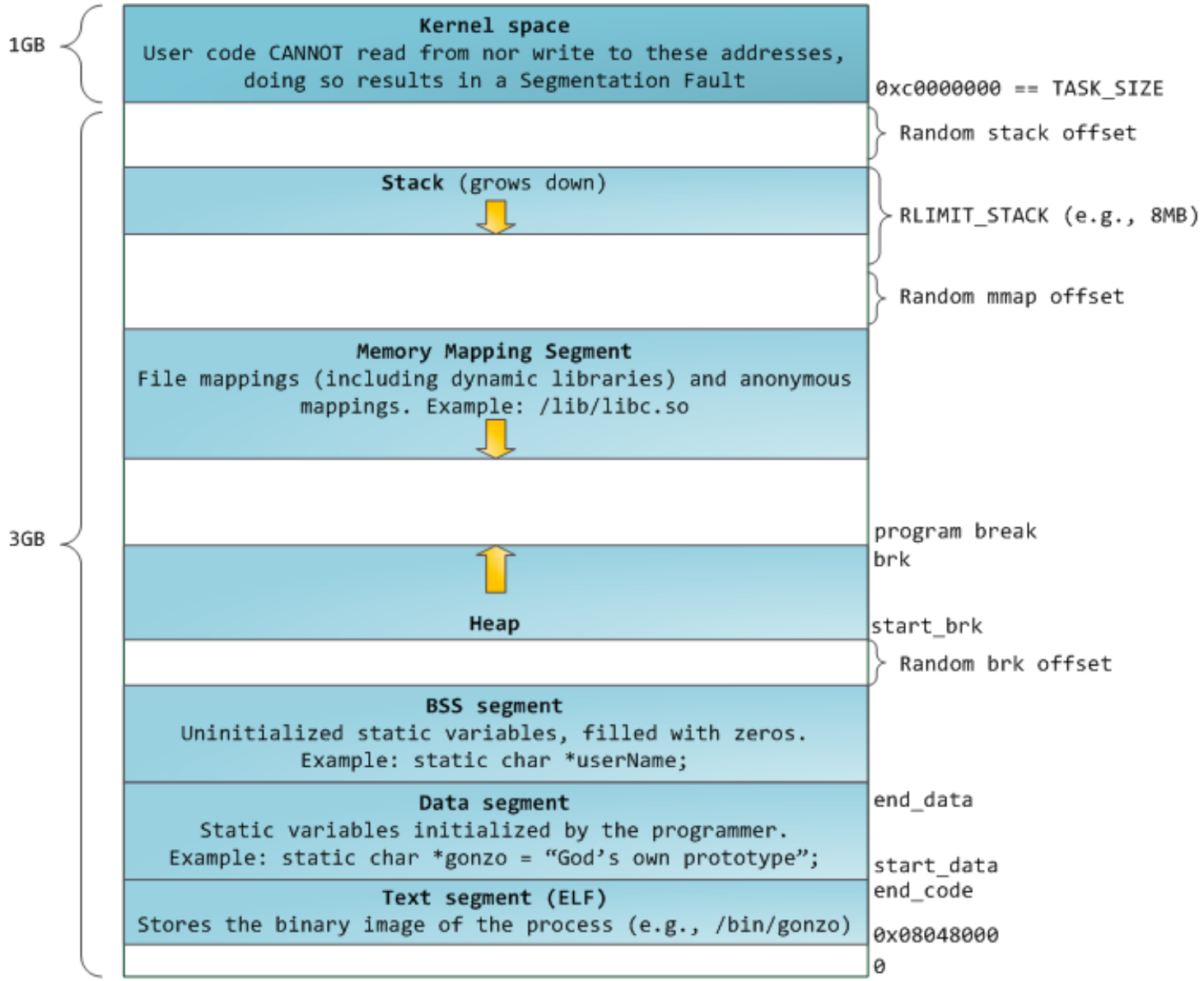
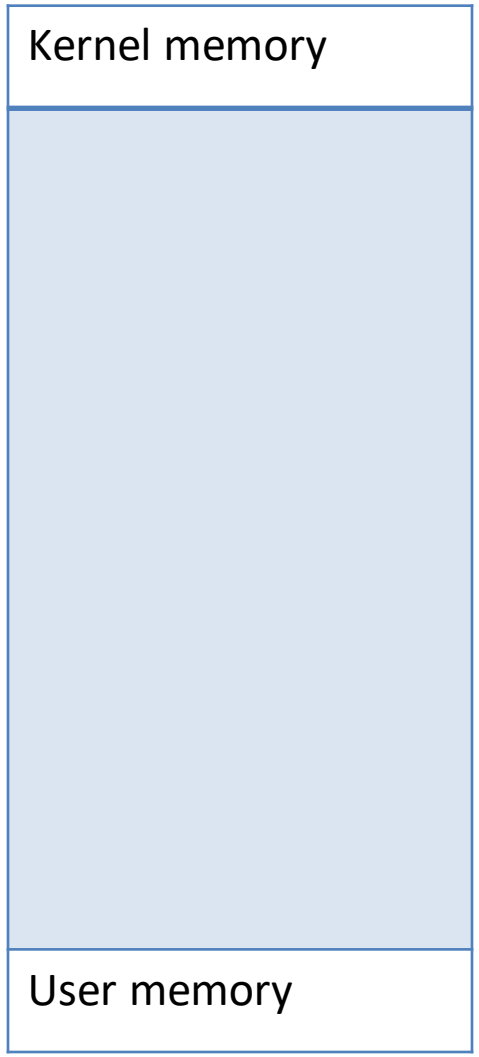


Image source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

x86-64 virtual address space (48 bits)



ffff_ffff_ffff_ffff

ffff_8000_0000_0000

0000_7fff_ffff_ffff

0000_0000_0000_0000

From top to bottom

128 TB of kernel-space virtual mem

Approx. 16 million TB unused space

128 TB of user-space virtual mem

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

Bits of randomness (x86-64)

- 64-bit has much more randomness than 32-bit x86
- Memory regions aligned on page boundaries have 28 bits of randomness
- Memory regions aligned on 2 MB boundaries (like programs and large libraries) have 19 bits of 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?

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