Secure Coding and Buffer Overflow Attacks

ECE344

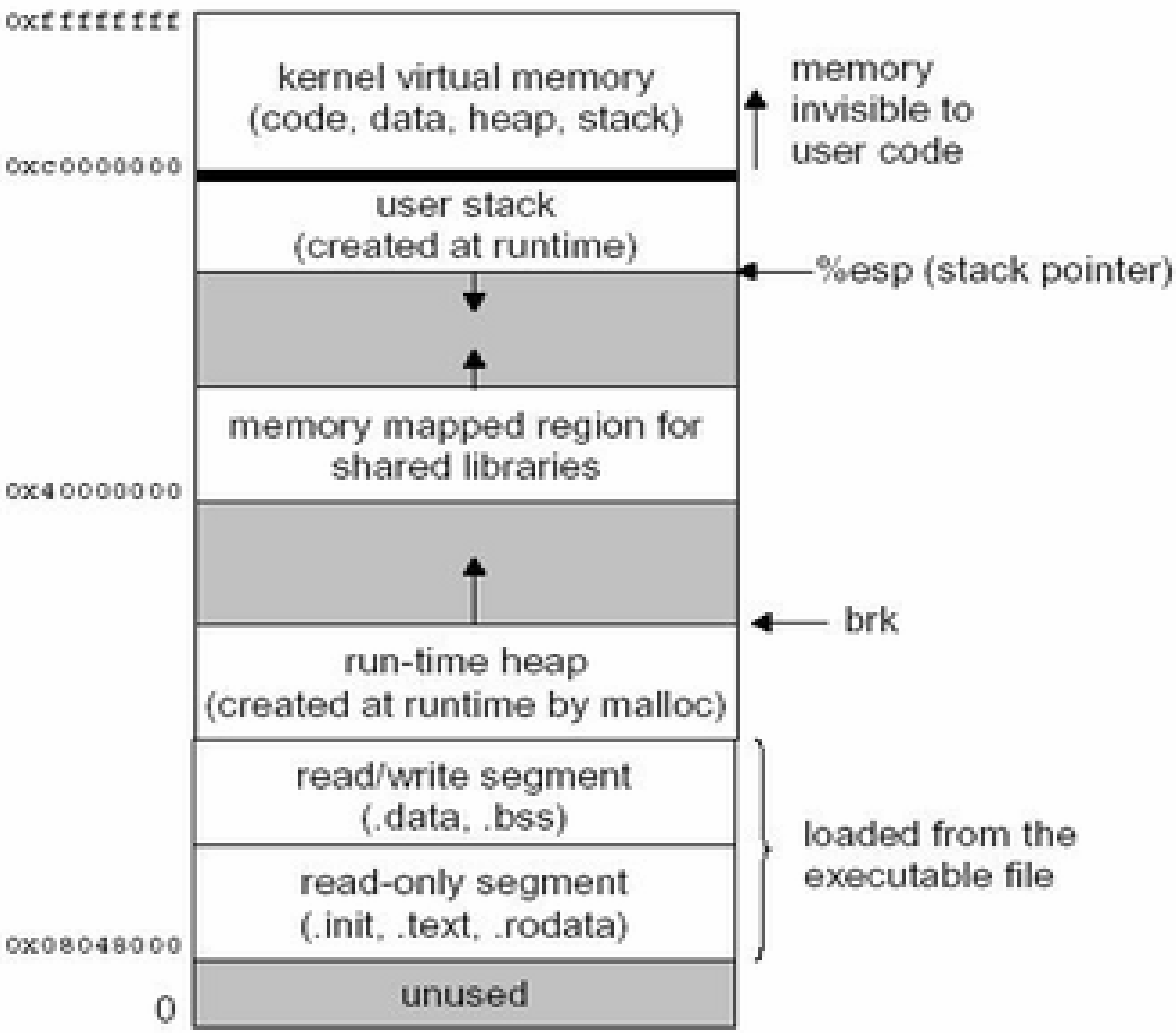
References

- **Book Chapter 19**
- **Smashing the Stack for Fun and Profit by Aleph One.**
- **Buffer Overflow Attacks and Their Countermeasures/** By Sandeep Grover.
- All examples in these slides draw for the above references
- Needless to say, this is for educational purposes only. I want you to understand weaknesses in systems and how to prevent introducing them ad developers.
- This may also serve you during an interview; writing safe code is a major requirement today.

Terminology

- Static variables are allocated at load time in the data segment
- Dynamic variables are allocated at run time on the stack
- Stack pointer points to the top of the stack
- Bottom of stack is at a fixed address
- Stack grows / shrinks dynamically

Process Layout

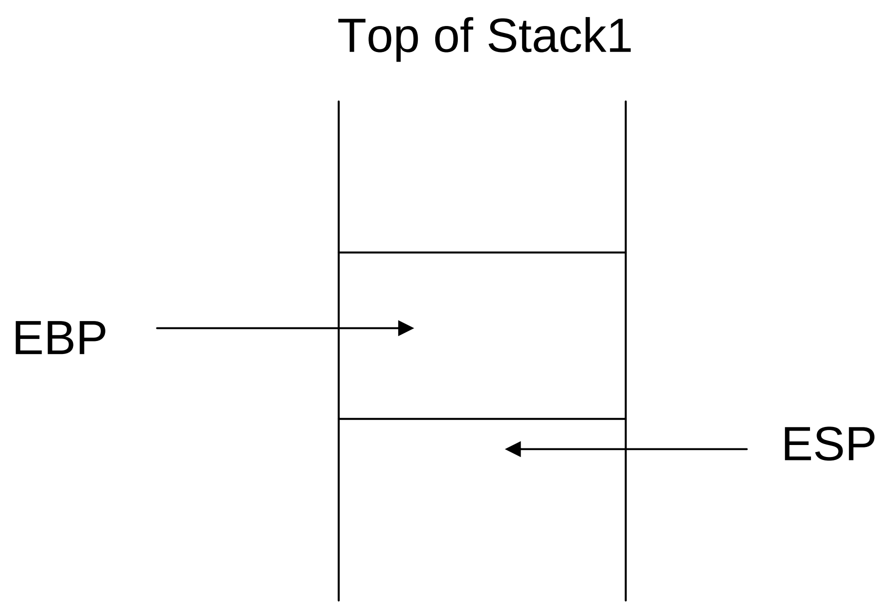


The Stack

- Stack is organized into logical stack frames
- Stack frames pushed and popped from stack as procedures are called and complete
- Stack frame
  - Parameters to a function
  - Function's local variables
  - Data to recover previous stack frame
  - Often a frame pointer
- Depending on the architecture stacks grow up or down
- Often stacks grow down (e.g., Motorola, Intel, Sparc, MIPS)

Stack Pointer and Frame Pointer

- EBP frame pointer
- ESP stack pointer



Example I

```
void function (int a, int b, int c) {
    char buffer1[5];
    char buffer2[10]; }

int main() {
    function(1,2,3);
}
```

c
b
a
ret
fp
…
…

```
$ gcc -S -o example1.s example1.c
pushl $3          // a onto stack
pushl $2          // b onto stack
pushl $1          // c onto stack
call function     // Pushes PC on stack

// Procedure prologue
pushl %ebp              // EBP onto stack
movl %esp,%ebp          // SP into EBP
subl $20,%esp           // Space for locals
```

Stack Layout for Example I

```
bottom of                                                   top of
memory                                                      memory
         buffer2 buffer1    sfp  ret   a     b     c
<------ [      ]  [         ] [   ] [   ] [   ] [   ] [   ]

top of                                                      bottom of
stack                                                       stack
```

- memory can only be addressed in word-size junks (4 bytes)
- buffer1 requires 8 bytes, i.e., 2 words
- buffer2 requires 12 bytes, i.e., 3 words

Buffer overflow

- Buffer overflow means to input more data into a buffer than it can handle
- i.e., to write beyond the limits of the buffer possibly overwriting what's there

Example II

```
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}
void main() {
    char large_string[256];
    int i;
    for(i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```

- strcpy copies the content of *str (i.e., large_string[]) to local function variable buffer[] until a null-character is found in large_string[]
- when run, this results in a segmentation faul
- Why?

What’s Happening?

- buffer is much smaller than what’s in *str

  16 vs. 256
- 250 bytes in the stack are overwritten with the content of large_string[]
- Including sfp, ret, *str …
- large_string contains ‘A’; in hex 0x41
- So the 4 bytes for ret contain
  - 0x41414141
- Which is outside the processes address space

Stack Layout for Example II

```
bottom of                                              top of
memory                                                 memory

      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…
       buffer        sfp   ret  *str
<------ [           ] [   ] [   ] [   ]

top of                                                 bottom of
stack                                                  stack
```

- This changes the flow of execution !!!

Exploit

- Point the procedure return address to a piece of executable code
- Place the executable code on the stack

Stack Layout

```
bottom of                                                   top of
memory                                                      memory

        buffer                   sfp      ret       *str
<------ [ssssssssssssssssss]  [ssss]  [0xD8]  [    ]

top of                                                      bottom of
stack                                                       stack
```

- Direct the return address to the beginning of the code on the stack

The Code To Execute

```
include <stdio.h>
void main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

In Assembly I

```
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to distribute …
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>: pushl %ebp
0x8000131 <main+1>: movl %esp,%ebp
0x8000133 <main+3>: subl $0x8,%esp
0x8000136 <main+6>: movl $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>: movl $0x0,0xfffffffc(%ebp)
…
```

In Assembly II

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
0x80002c0 <__execve+4>: movl $0xb,%eax
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
0x80002c8 <__execve+12>: movl0xc(%ebp),%ecx
…
```

Addresses

- We need “/bin/sh” as string in memory
- We need it’s address

- We don’t know what the address will be at run time

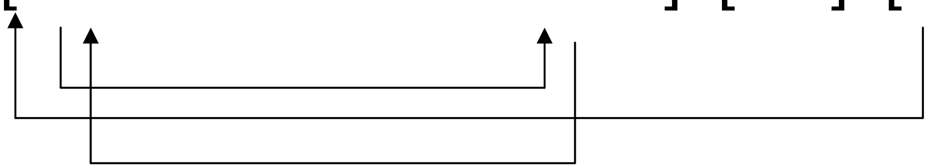
- So, use, PC relative addressing, i.e., address a location as an offset relative to PC

Stack Layout

bottom of memory — top of memory

buffer — sfp — ret — …

<------ [JJSSSSSSSSSSSCCss] [sss] [0xD8] [ ]



top of stack — bottom of stack

•

- S is the shell code — JJ is a JMP instruction
- s is the “/bin/sh” string — CC is a call instruction
- JMP / CALL can use PC relative addressing

More Details

- There are a few more details to get this to work
- Translate the exploitation code into assembly
- Calculate the relative address of JMP and CALL
- Make sure that the final code does not contain any NULL characters, as the exploit is injected as a string
- …

Counter-measures

- Write secure code. Careful
  - strcpy, strcat, sprintf, vsprintf operate on NULL-terminated strings without bounds checking
  - gets reads input until EOF
  - scanf
- Do bounds checking in code yourself
- Use strncpy *et al.*

Counter-measures

- Disallow executing code on stack
  - Sometimes difficult, as part of compiler design
  - Possible in Linux
  - Possible in newer versions of Solaris with support from architecture (Sparc)
- Compiler tools and compiler warnings
  - Detect unsafe code and warns user
  - Do not ignore compiler warnings

Counter-measures

- Look at
  - Libsafe
  - Safeguard
  - Stackshield