

Buffer Overflows

- Technique to force execution of malicious code with unauthorized privileges
 - launch a command shell
 - search local disk or network for sensitive data
 - register with command and control network as a zombie
- Can be applied both locally and remotely
- Attack technique is independent of machine architecture and operating system
- Can be tricky to execute, but extremely effective

Definitions

Buffer: a contiguous block of computer memory that holds multiple instances of the same type (C arrays)

Overflow: to fill over the brim, to fill more than full

Buffer Overflow: happens when a program attempts to write data outside of the memory allocated for that data

- Usually affects buffers of fixed size
- Also known as **Buffer Overrun**

Simple Example

Off-by-one errors are common and can be exploitable! (see Phrack 55)

```
char B[10];
```

```
B[10] = x;
```

- Array starts at index zero
- So [10] is 11th element
- One byte outside buffer was referenced

Another Example

```
function foo(char * a) {  
    char b[100];  
    ...  
    strcpy(b, a); // (dest, source)  
    ...  
}
```

- What is the size of the string located at "a"?
- Is it even a null-terminated string?
- What if it was "strcpy(a, b);" instead?
 - What is the size of the buffer pointed to by "a"?

What Happens When Memory Outside a Buffer Is Accessed?

- If memory doesn't exist:
 - Bus error
- If memory protection denies access:
 - Segmentation fault
 - General protection fault
- If access is allowed, memory next to the buffer can be accessed
 - Heap
 - Stack
 - Etc...

Real Example: efingerd.c, v. 1.5

- CAN-2002-0423

```
static char *lookup_addr(struct in_addr
in) {
    static char addr[100];
    struct hostent *he;
    he = gethostbyaddr(...)
    strcpy (addr, he->h_name);
    return addr;
}
```

- How big is `he->h_name`?
- Who controls the results of `gethostbyaddr`?
- How secure is DNS? Can you be tricked into looking up a maliciously engineered value?

Fundamental "C" Problems

- You can't know the length of buffers just from a pointer
 - Partial solution: pass the length as a separate argument
- "C" string functions aren't safe
 - No guarantees that the new string will be null-terminated!
 - Doing all checks completely and properly is tedious and tricky

“Overflowing” Functions

- `gets()`
 - ```
void main() {
 char buf[512];
 gets(buf);
}
```
- `strcpy()`, `strcat()`
  - ```
int main(int argc, char ** argv) {  
    char buf[512];  
    strcpy(buf, argv[1]);  
}
```
- `sprintf()`, `vsprintf()`, `scanf()`, `sscanf()`, `fscanf()`
- and also your own custom input routines...

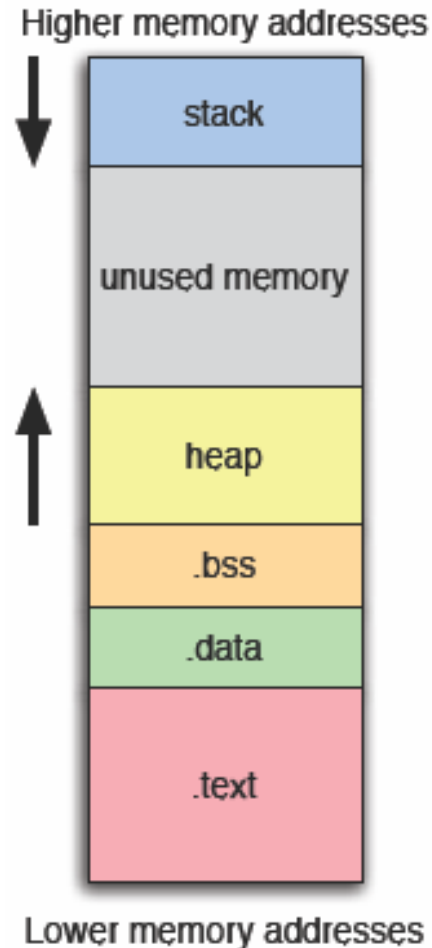
Process Memory Organization

- Text section (.text)
 - Includes instructions and read-only data
 - Usually marked read-only
 - Modifications cause segment faults
- Data section (.data, .bss)
 - Initialized and uninitialized data
 - Static variables
 - Global variables

Process Memory Organization

- Stack section
 - Used for implementing procedure abstraction
- Heap section
 - Used for dynamically allocated data
- Environment/Argument section
 - Used for environment data
 - Used for the command line data

Linux x86 Process Layout



- Process memory partitioned into segments
 - `.text` Program code
 - `.data` Initialized static data
 - `.bss` Uninitialized static data
 - `heap` Dynamically-allocated memory
 - `stack` Program call stack
- Each memory segment has a set of permissions associated with it
 - Read, write, and execute (rwx)

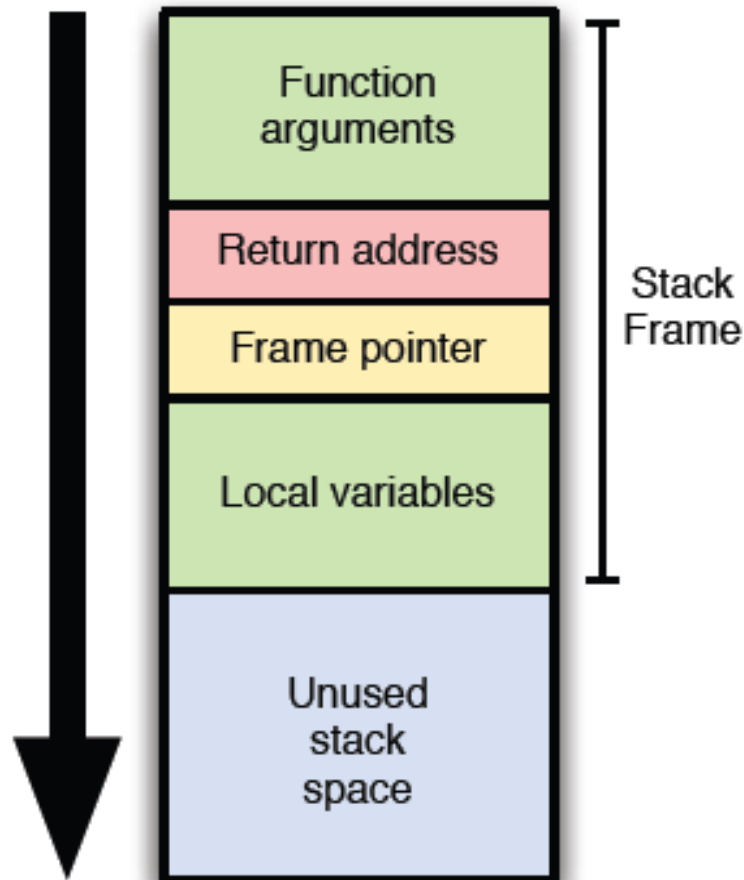
The Stack

- The stack usually grows towards lower memory addresses
- This is the way the stack grows on many architectures including the Intel, Motorola, SPARC, and MIPS processors
- The stack pointer (SP) points to the top of the stack (usually last valid address)

Frame Structure

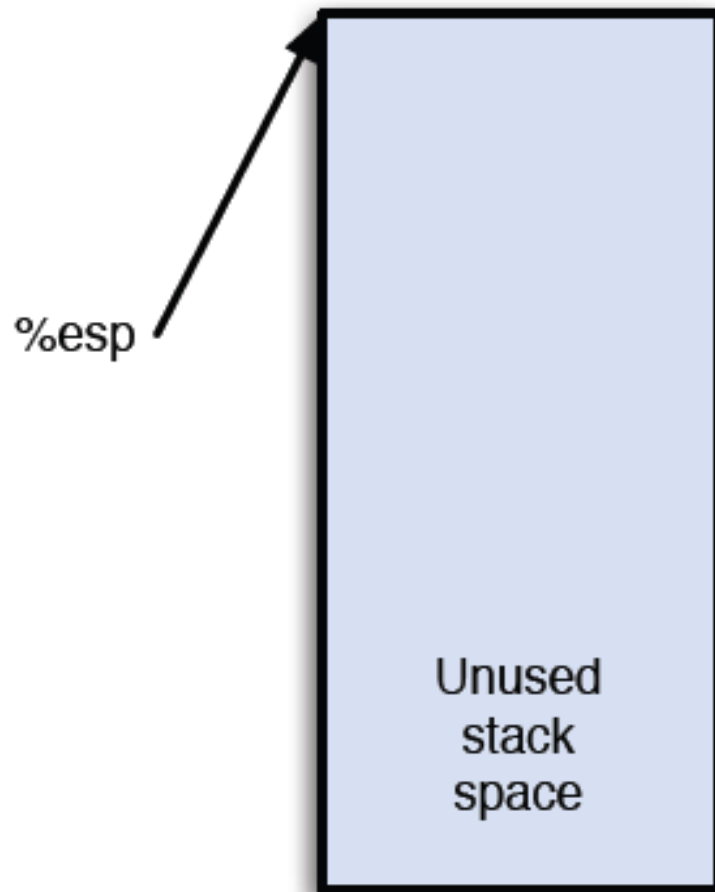
- The stack is composed of frames
- Frames are pushed on the stack as a consequence of function calls (function prolog)
- The address of the current frame is stored in the Frame Pointer (FP) register
 - On Intel architectures EBP is used for this purpose
- Each frame contains
 - The function's actual parameters
 - The return address to jump to at the end of the function
 - The pointer to the previous frame
 - Function's local variables

Structure of the ix86 Stack



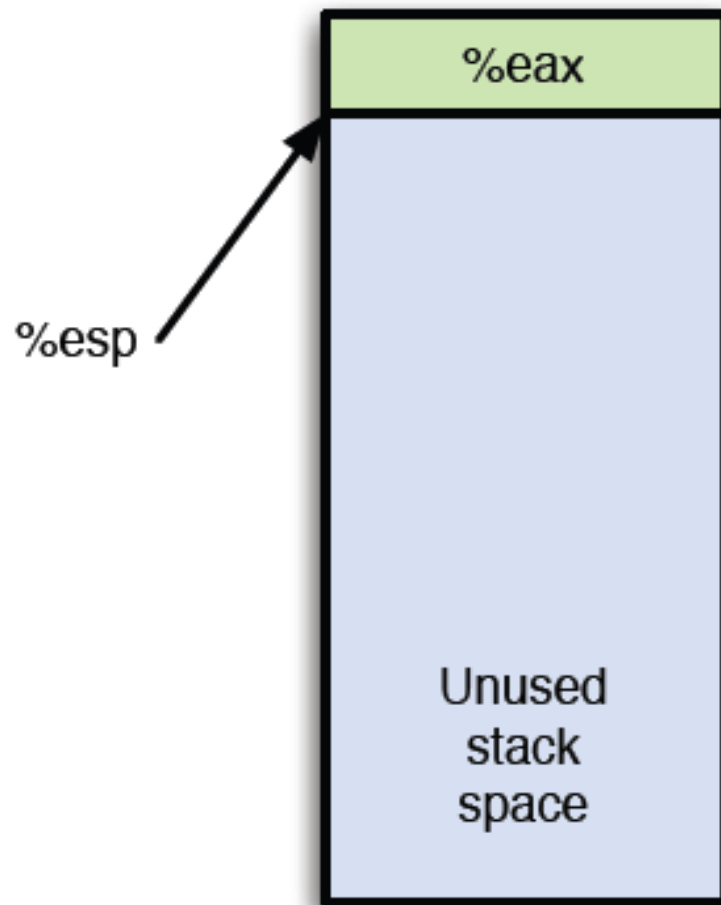
- Used to implement procedure abstraction
- Stack composed of frames, each of which corresponds to a unique function invocation
 - function arguments
 - return address (**eip**)
 - frame pointer (**ebp**)
 - local “automatic” data
- Grows downward from higher to lower memory addresses

Stack Frame Setup and Teardown



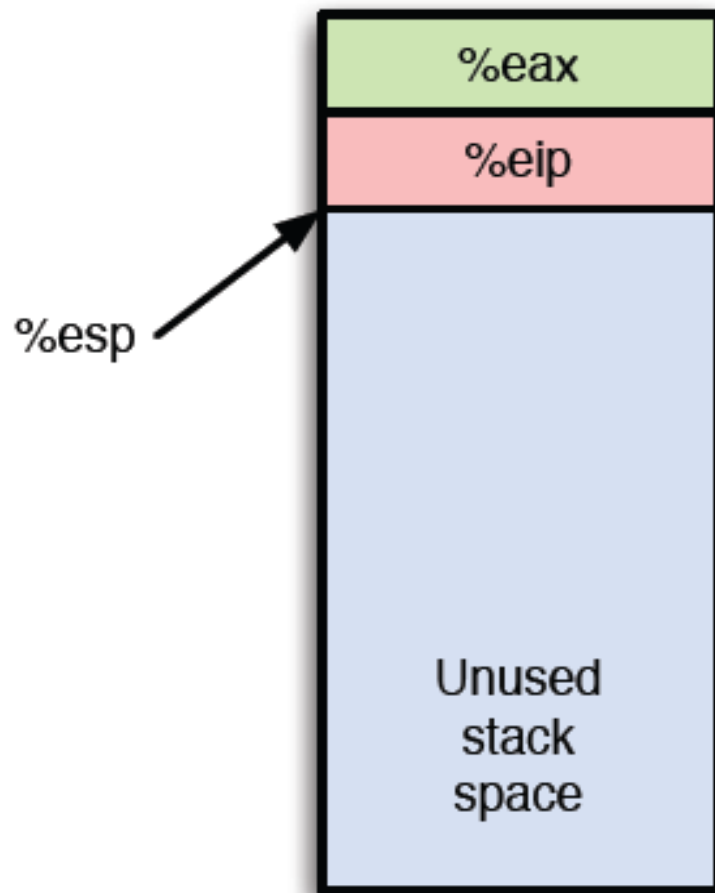
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



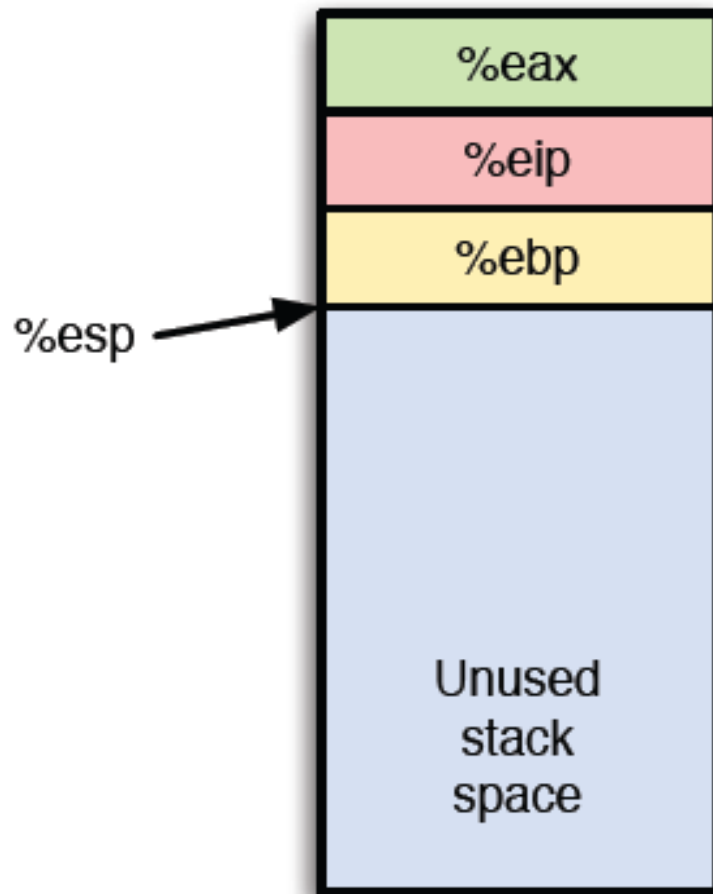
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



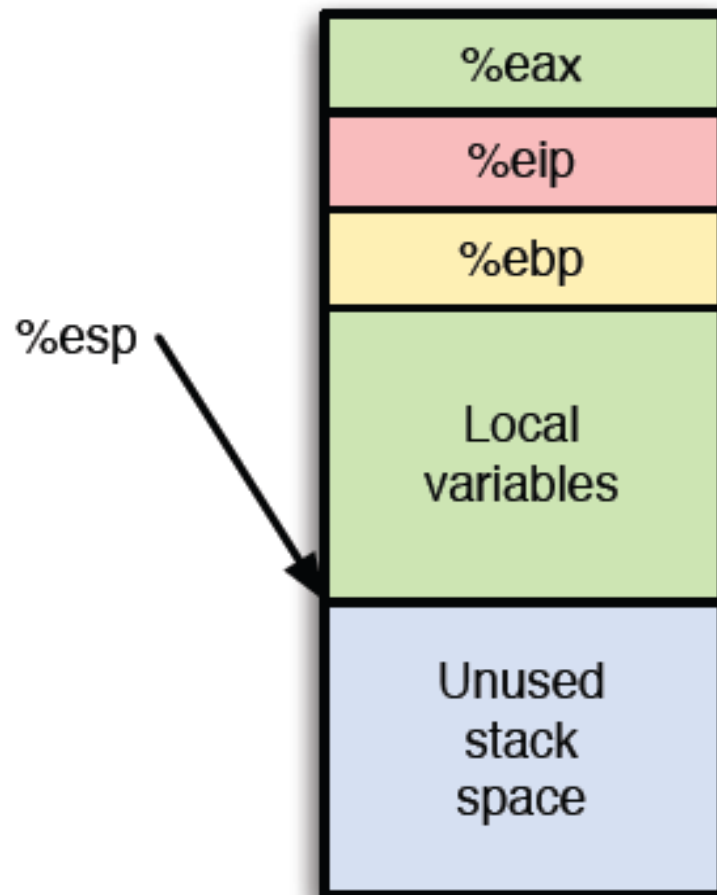
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



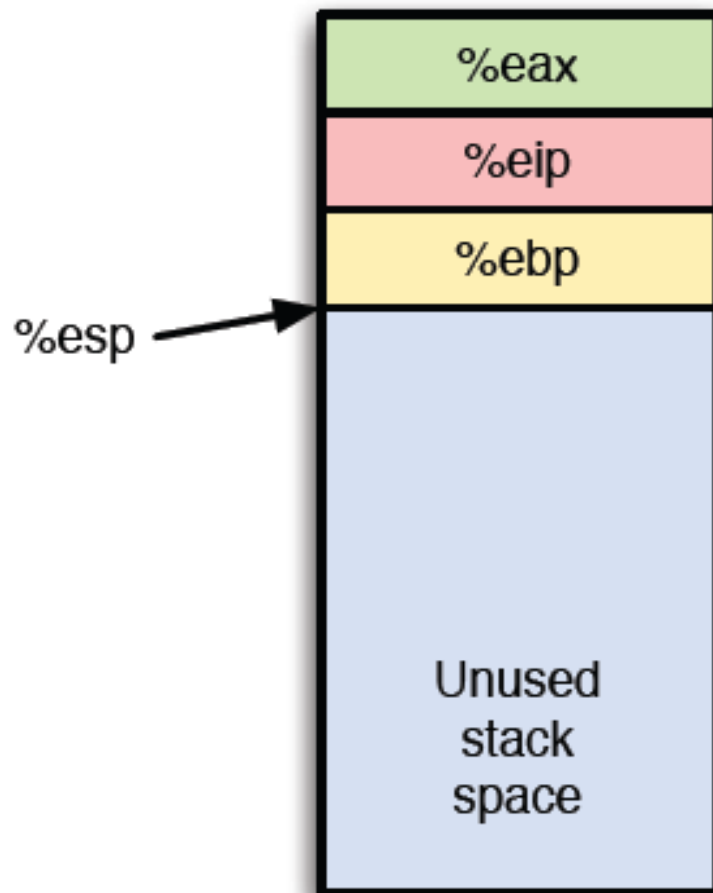
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



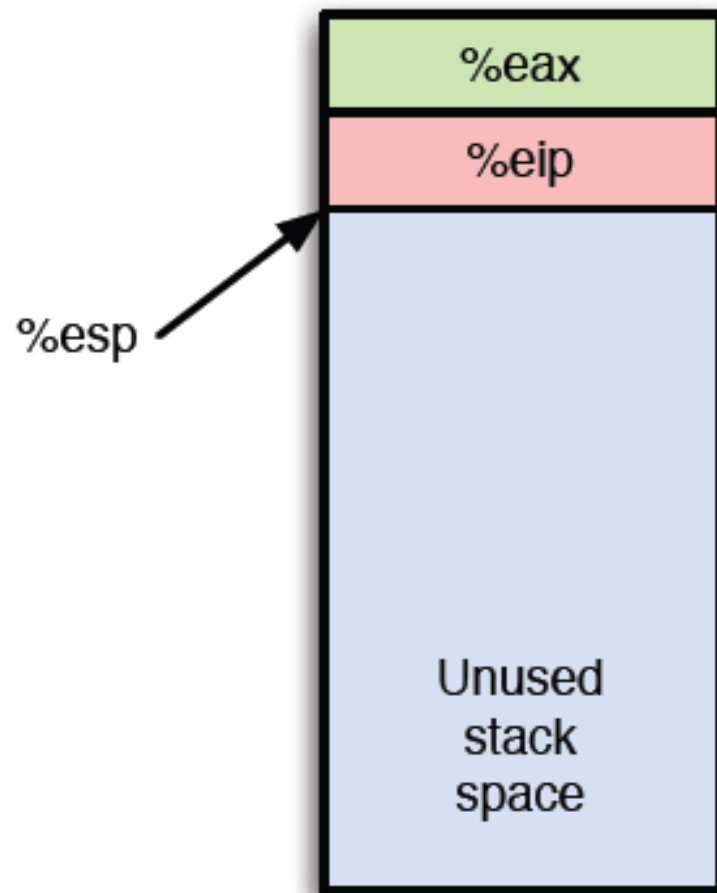
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



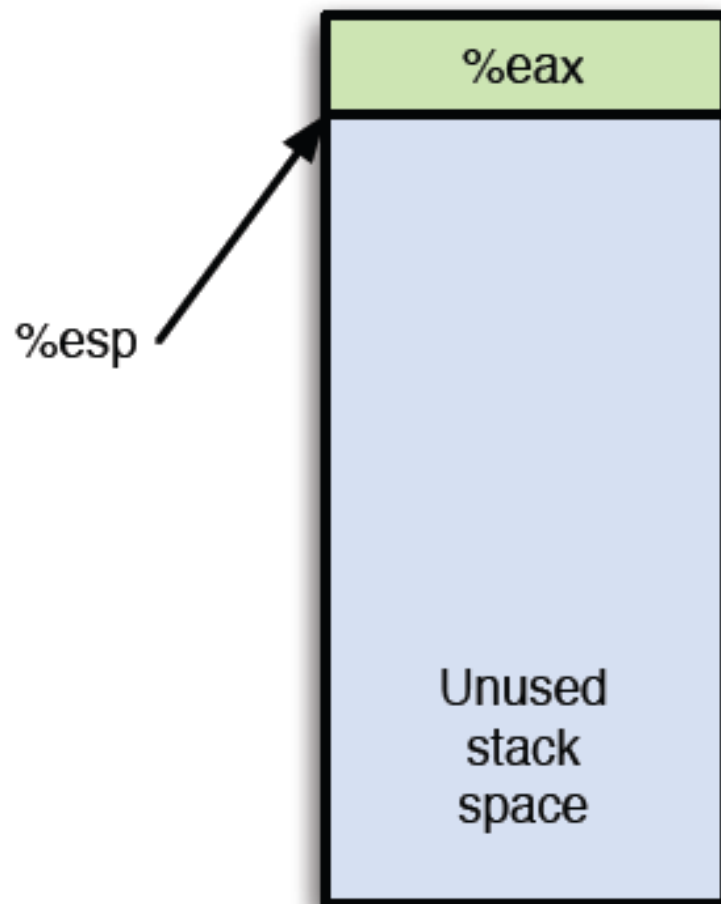
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



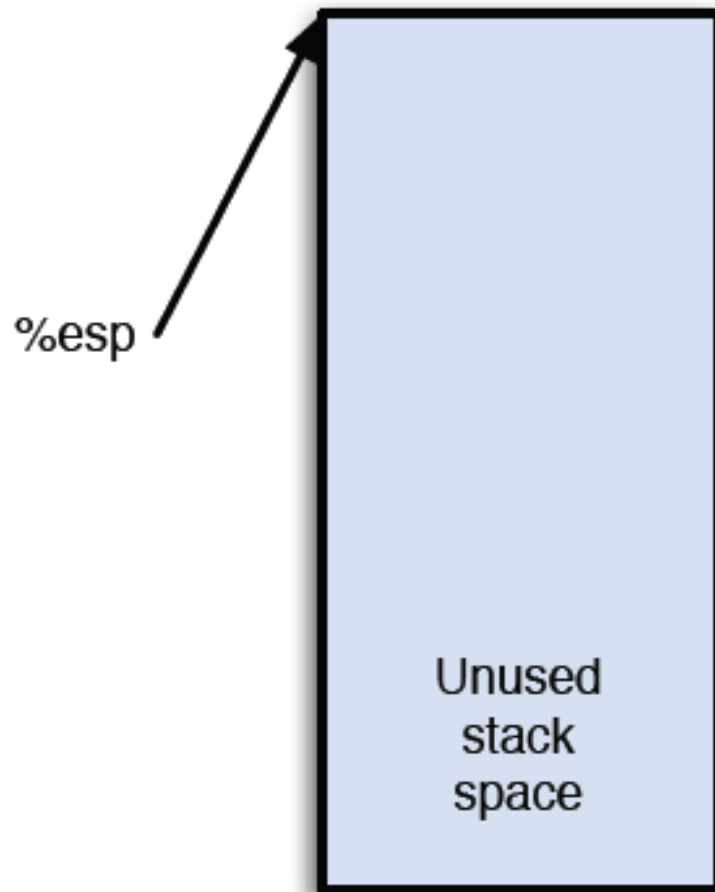
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



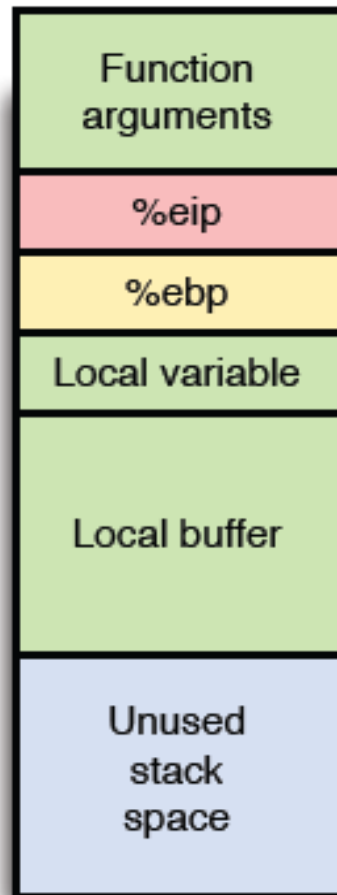
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Stack Frame Setup and Teardown



```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

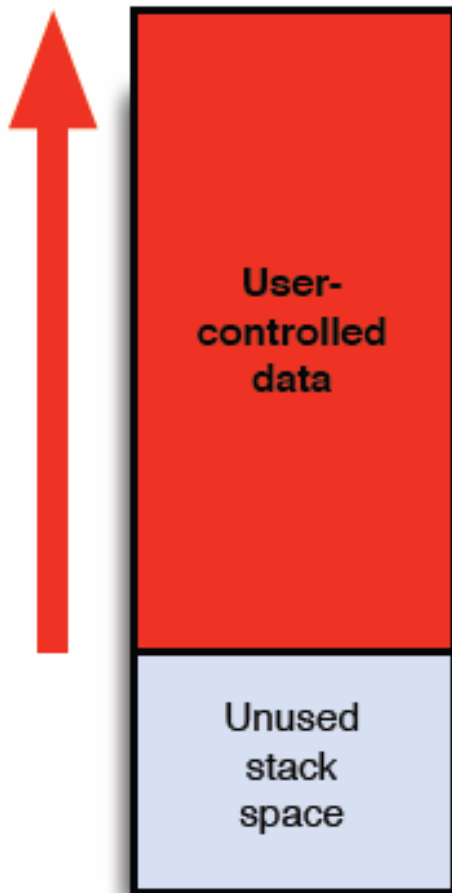
Vulnerability of Stack Structure



A **small problem**: return address is inlined with user-controlled buffers

- What can happen if copy into stack-allocated buffer is not bounds-checked?

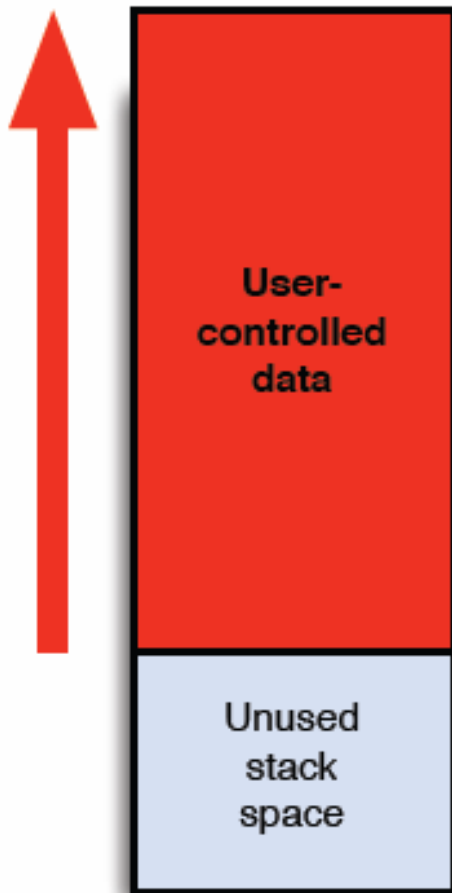
Vulnerability of Stack Structure



A **small problem**: return address is inlined with user-controlled buffers

- What can happen if copy into stack-allocated buffer is not bounds-checked?
- User can control values of other variables, frame pointer, and *return address*
- If user overwrites the return address on stack, what happens when function returns?

Vulnerability of Stack Structure



A small problem: return address is inlined with user-controlled buffers

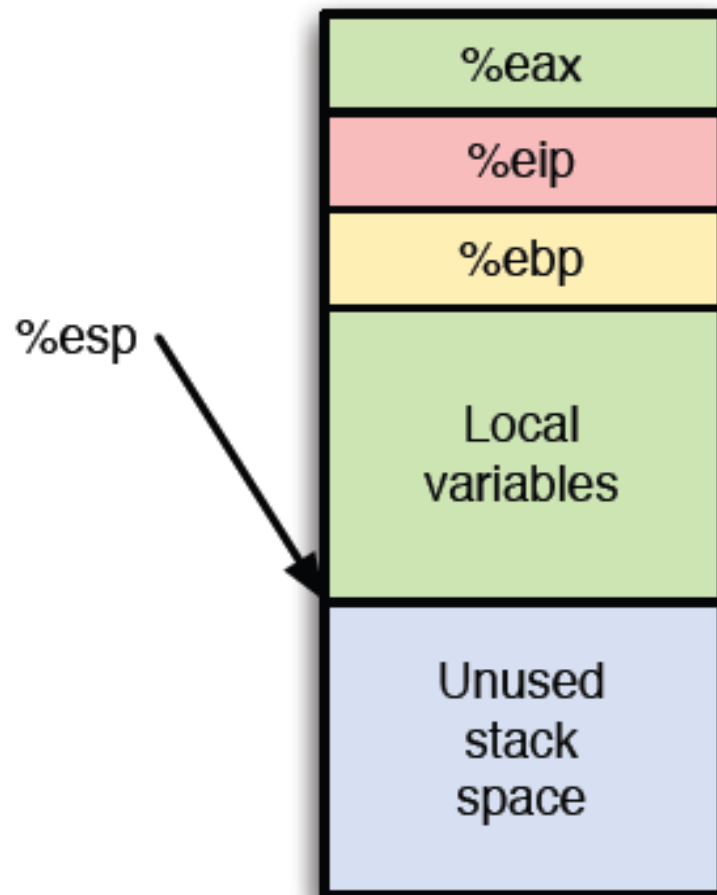
- What can happen if copy into stack-allocated buffer is not bounds-checked?
- User can control values of other variables, frame pointer, and return address
- If user overwrites the return address on stack, what happens when function returns

Result: process will execute arbitrary code of the user's choosing

Side Effects of Buffer Overflow Depend On

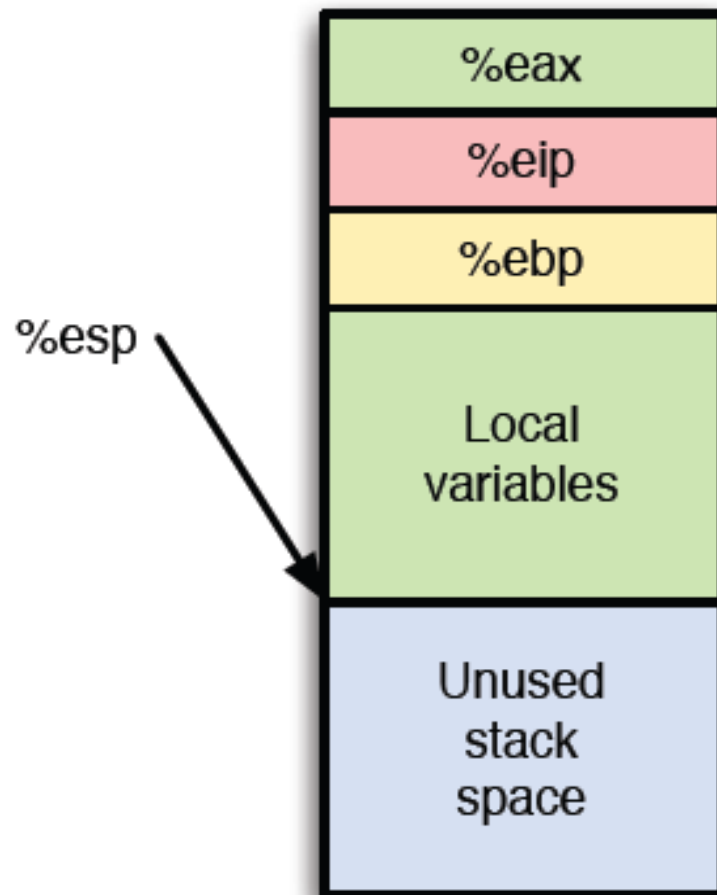
- How much data written past the bounds
- What data is overwritten
- Whether the program attempts to read the data overwritten
- What data replaces the memory that gets overwritten

Smashing the Stack



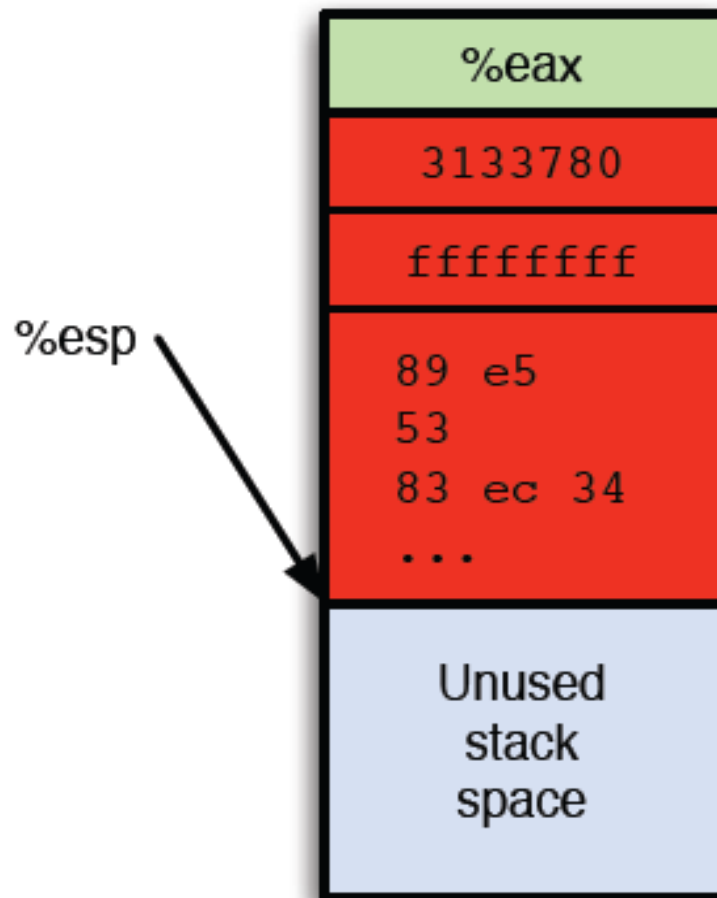
```
8048716 mov  %eax,(%esp)
8048719 call 80485ed <do chksum>
80485ed push %ebp
80485ee mov  %esp,%ebp
80485f1 sub  $0x34,%esp
...
804866c add  $0x34,%esp
8048670 pop  %ebp
8048671 ret
```

Smashing the Stack



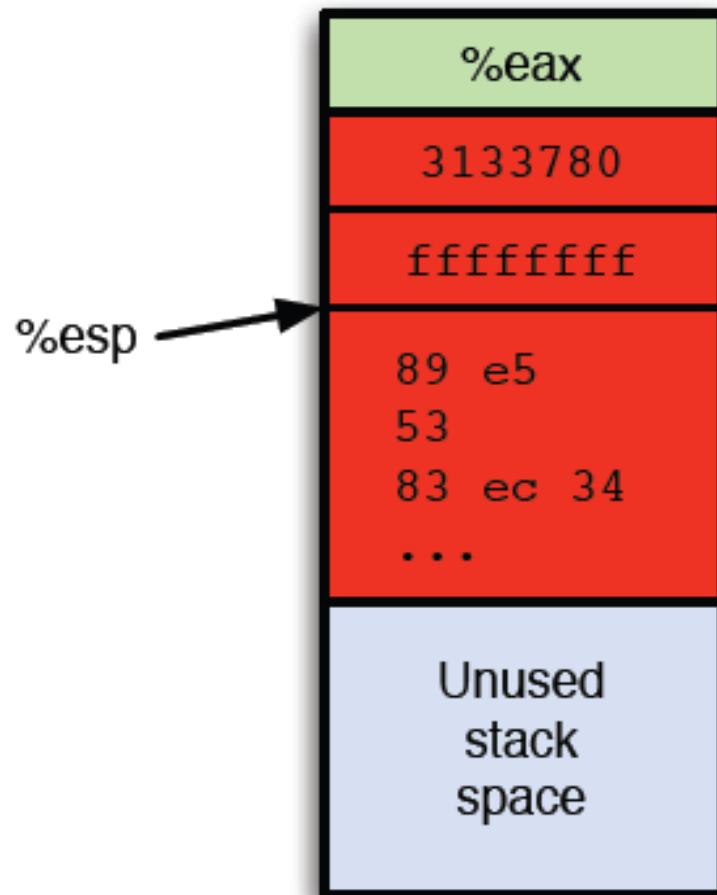
```
...  
8048624 mov 0x8(%ebp),%eax  
8048627 mov %eax,0x4(%esp)  
804862b lea 0xfffffe4(%ebp),%eax  
804862e mov %eax,(%esp)  
8048631 call 80483f8 <strcpy@plt>  
  
...  
804866c add $0x34,%esp  
8048670 pop %ebp  
8048671 ret
```

Smashing the Stack



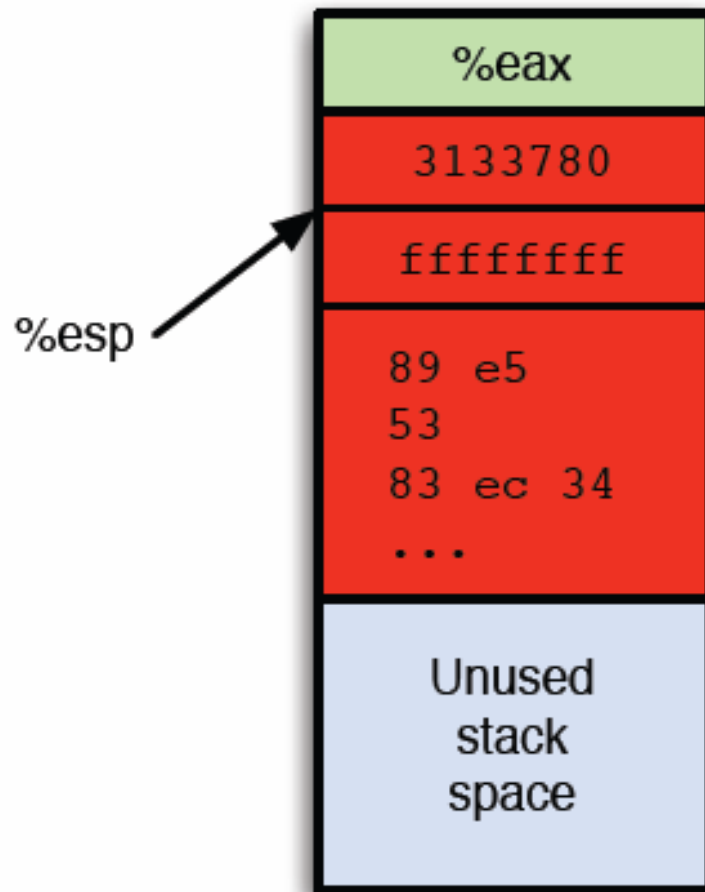
```
...  
8048624 mov 0x8(%ebp),%eax  
8048627 mov %eax,0x4(%esp)  
804862b lea 0xfffffe4(%ebp),%eax  
804862e mov %eax,(%esp)  
8048631 call 80483f8 <strcpy@plt>  
...  
804866c add $0x34,%esp  
8048670 pop %ebp  
8048671 ret
```

Smashing the Stack



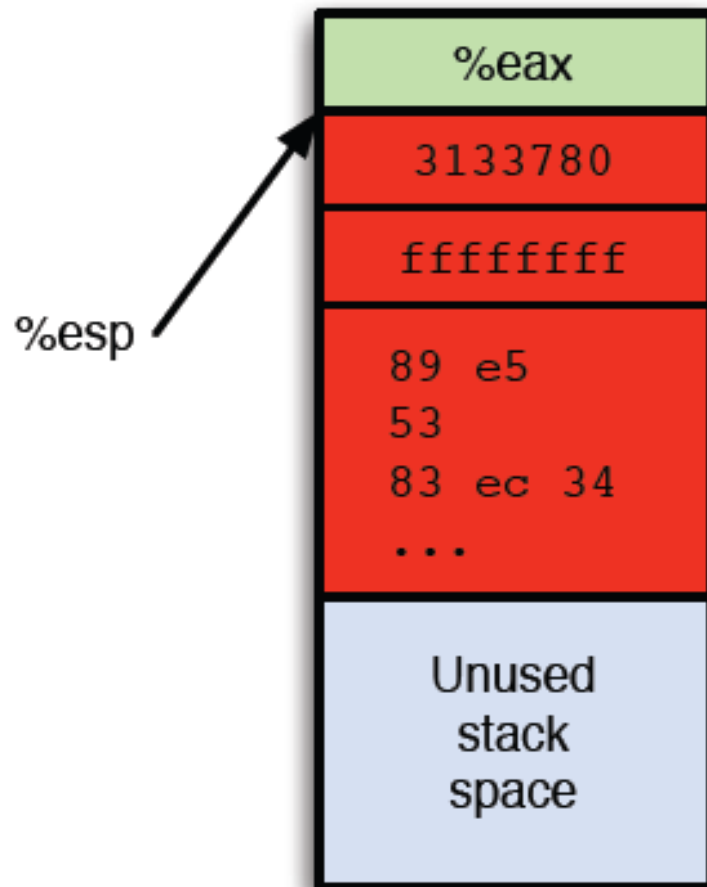
```
...
8048624 mov 0x8(%ebp),%eax
8048627 mov %eax,0x4(%esp)
804862b lea 0xfffffe4(%ebp),%eax
804862e mov %eax,(%esp)
8048631 call 80483f8 <strcpy@plt>
...
804866c add $0x34,%esp
8048670 pop %ebp
8048671 ret
```

Smashing the Stack



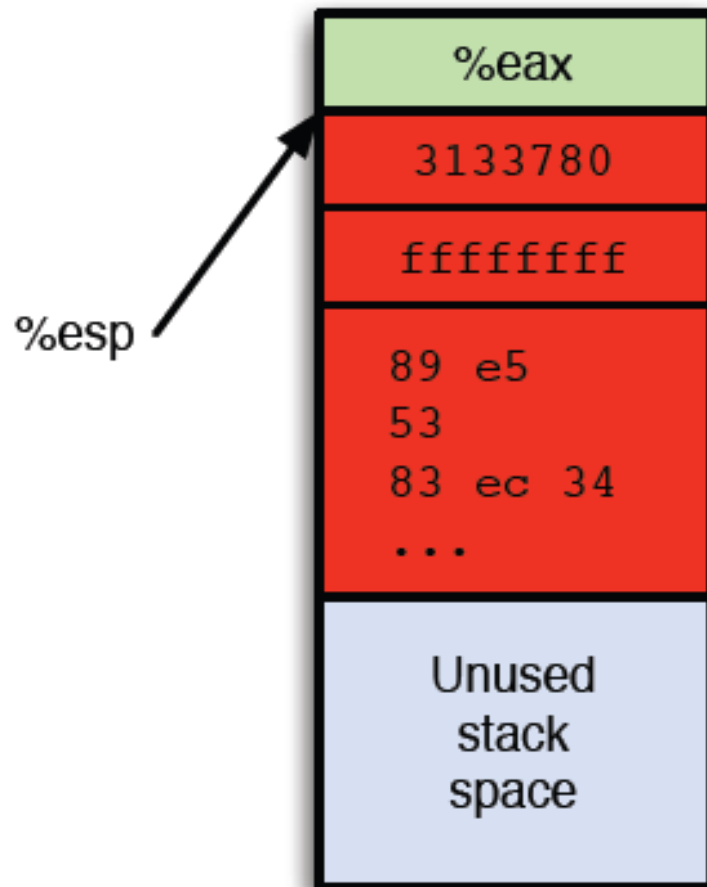
```
...
8048624  mov  0x8(%ebp),%eax
8048627  mov  %eax,0x4(%esp)
804862b  lea  0xfffffe4(%ebp),%eax
804862e  mov  %eax,(%esp)
8048631  call 80483f8 <strcpy@plt>
...
804866c  add  $0x34,%esp
8048670  pop  %ebp
8048671  ret
```

Smashing the Stack



```
...  
8048624 mov 0x8(%ebp),%eax  
8048627 mov %eax,0x4(%esp)  
804862b lea 0xfffffe4(%ebp),%eax  
804862e mov %eax,(%esp)  
8048631 call 80483f8 <strcpy@plt>  
...  
804866c add $0x34,%esp  
8048670 pop %ebp  
8048671 ret
```

Smashing the Stack



```
...
8048624 mov 0x8(%ebp),%eax
8048627 mov %eax,0x4(%esp)
804862b lea 0xfffffe4(%ebp),%eax
804862e mov %eax,(%esp)
8048631 call 80483f8 <strcpy@plt>
...
804866c add $0x34,%esp
8048670 pop %ebp
8048671 ret
3133780 xor %eax,%eax
```

Memory Layout for Frame

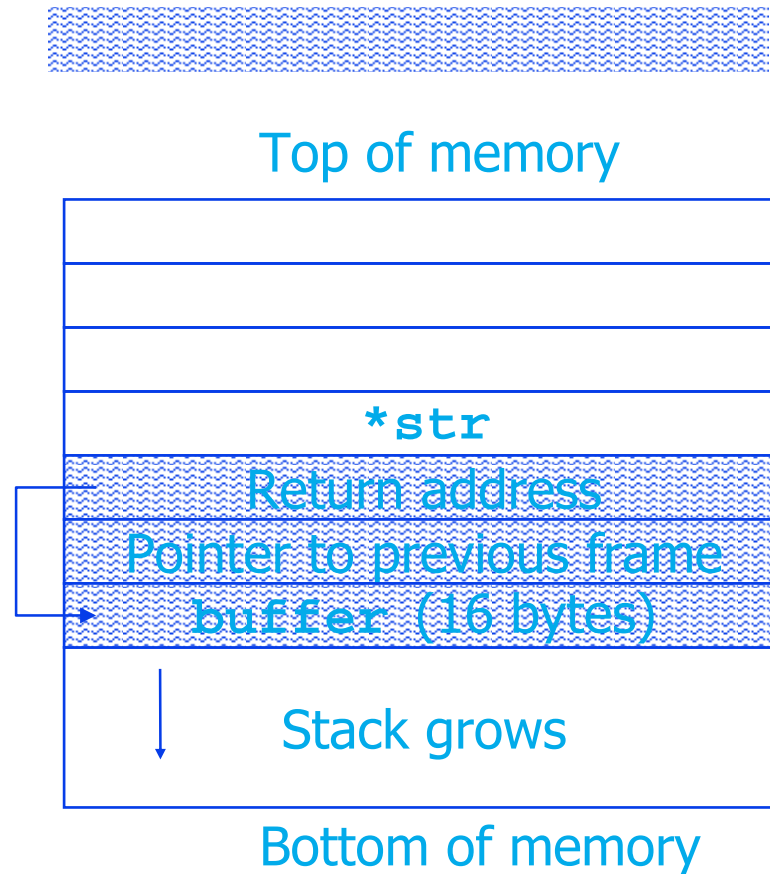


Buffer Overflow

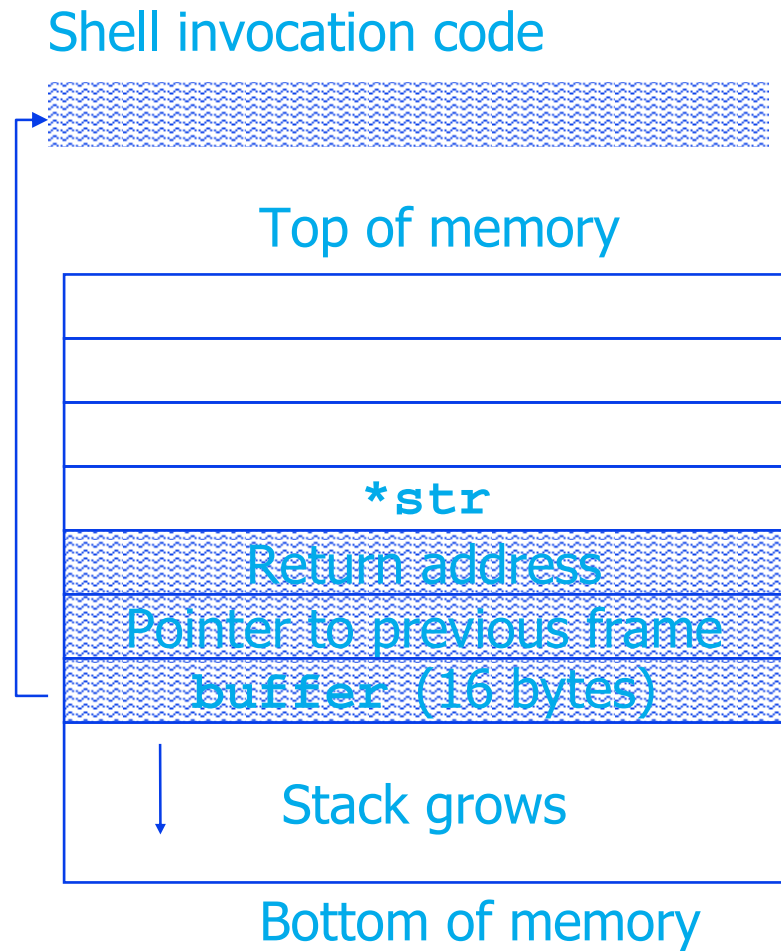
- Data is copied without checking boundaries
- Data “overflows” a pre-allocated buffer and overwrites the return address
- Normally this causes a segmentation fault
- If correctly crafted, it is possible overwrite the return address with a user-defined value
- It is possible to cause a jump to user-defined code (e.g., code that invokes a shell)
- The code may be part of the overflowing data (or not)
- The code will be executed with the privileges of the running program

Buffer Overflow

Shell invocation code



Buffer Overflow



How to Exploit a Buffer Overflow

- Different variations to accommodate different architectures
 - Assembly instructions
 - Operating system calls
 - Alignment
- Linux buffer overflows explained in the paper “*Smashing The Stack For Fun And Profit*” by Aleph One, published on Phrack Magazine, 49(7)
- Most difficult task: generate the correct “payload”

The Shell Code

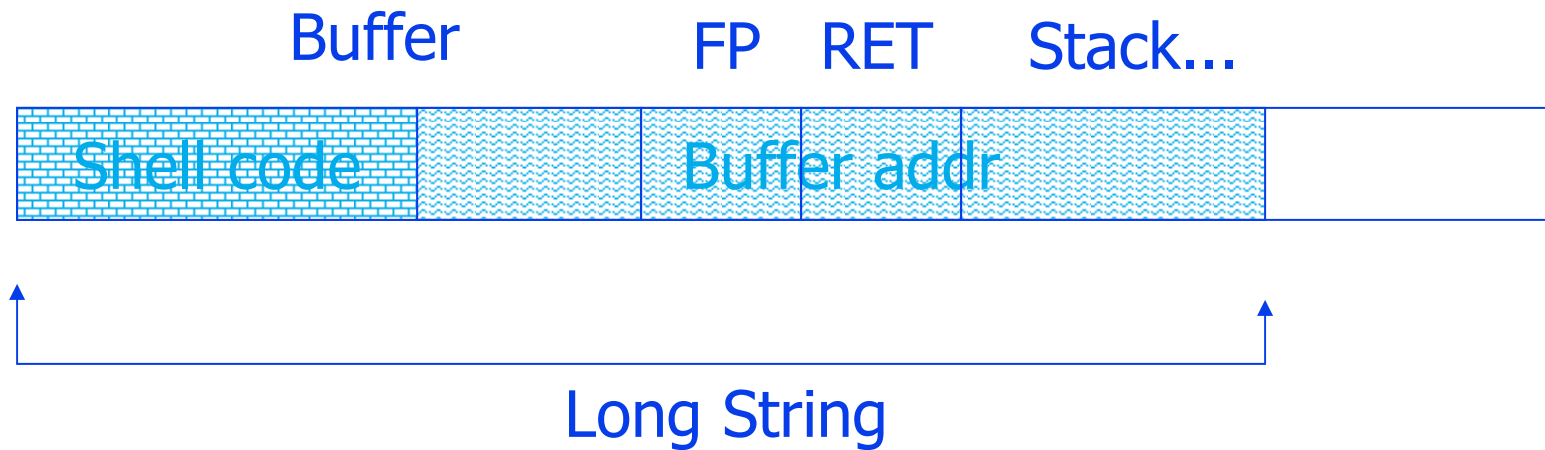
```
void main() {  
    char *name[2];  
  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
    exit(0);  
}
```

- System calls in assembly are invoked by saving parameters either on the stack or in registers and then calling the software interrupt (0x80 in linux)

High Level View

- Compile attack code
- Extract the binary for the piece that actually does the work
- Insert the compiled code into the buffer
 - Before or after the return address
- Figure out where overflow code should jump
- Place that address in the buffer at the proper location so that the normal return address gets overwritten

Executing the Shell Code



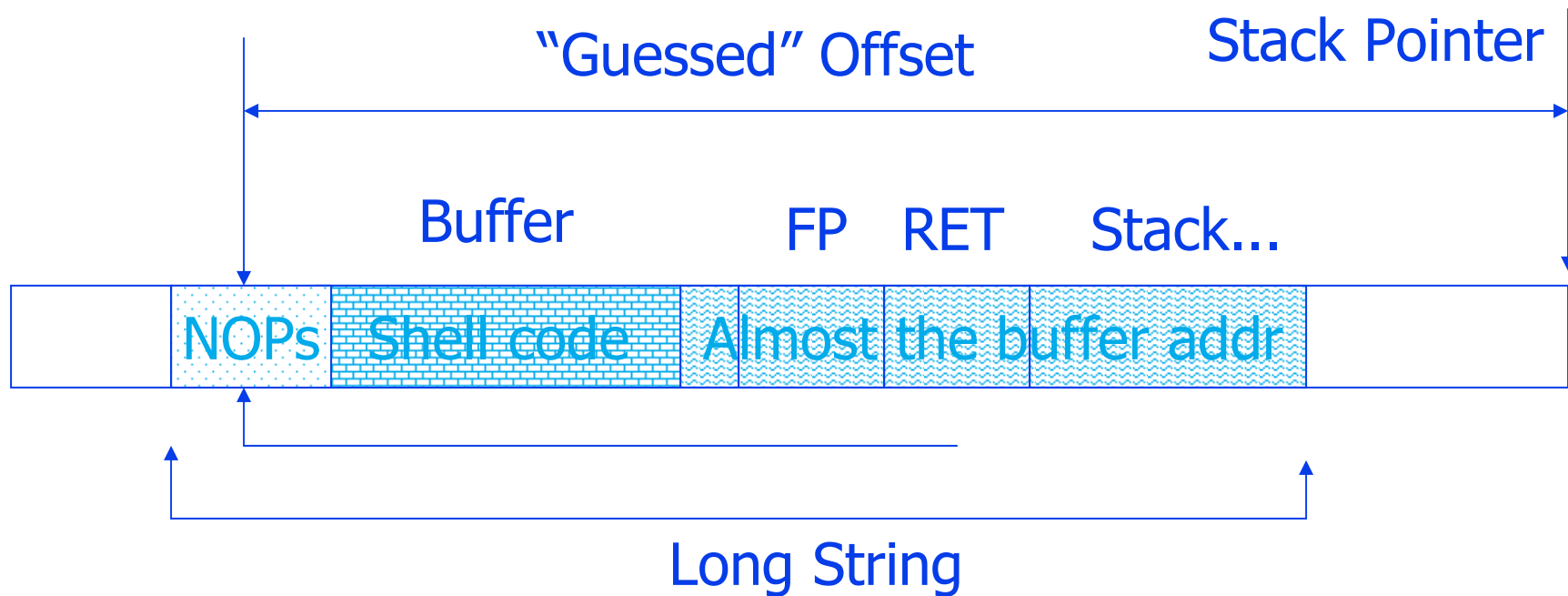
Guessing the Buffer Address

- In most cases the address of the buffer is not known
- It has to be “guessed” (and the guess must be very precise)
- Given the same environment and knowing the size of command-line arguments the address of the stack can be roughly guessed
- The stack address of a program can be obtained by using the function

```
unsigned long get_sp(void) {  
    __asm__( "movl %esp,%eax" );  
}
```
- We also have to guess the offset of the buffer with respect to the stack pointer

NOP Sled

Use a series of NOPs at the beginning of the overflowing buffer so that the jump does not need to be too precise (aka no-operation sled)



Heap Overflows

- Overflowing dynamically allocated (heap) buffers may overwrite malloc's “bookkeeping” structs
- Example struct from dlmalloc

```
struct malloc_chunk {  
    INTERNAL_SIZE_T    prev_size;  
    INTERNAL_SIZE_T    size;  
    struct malloc_chunk *bk;  
    struct malloc_chunk *fd;  
};
```

Other Buffer Overflows

- Return into libc (control is passed to library call instead of shell code, e.g., `system()`)
- Dtor overflow (C “global” destructor function override)
- C++ VPTR overflows (overwriting C++ virtual function pointers)

Remote Buffer Overflows

- Buffer overflow in a network server program can be exercised by an outside user
- Often provides the attacker with an interactive shell on the machine
 - Resulting session has the privileges of the process running the compromised network service
- One of the most common techniques to get remote access to a system

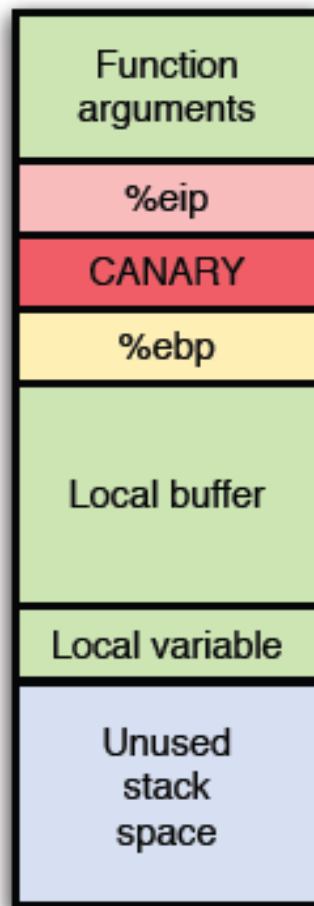
Solutions to Buffer Overflows

- Write decent programs
- Use a language that performs boundary checking (e.g., Java, C#, Python)
- Use Libsafe as a replacement for dangerous functions
- Use fgets, snprintf, strncat, strncpy, ...
- Use of canary values on function frames
- Make the stack non-executable (e.g., OpenWall project). This may solve some of the problems but not all of them
- Misuse-based intrusion detection

Canaries on a Stack

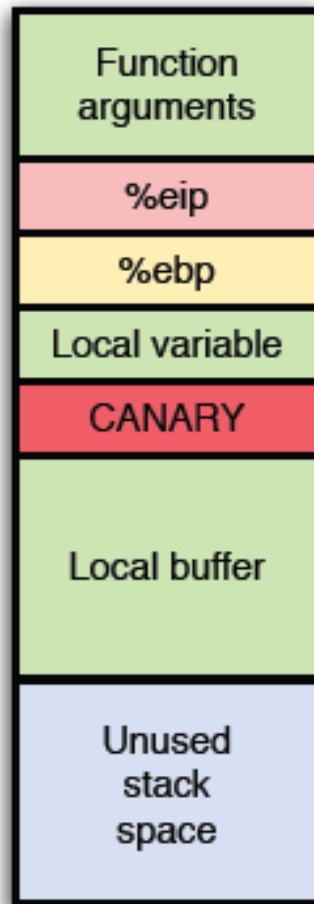
- Add a few bytes containing special values between variables on the stack and the return address.
- Before the function returns, check that the values are intact.
 - If not, there has been a buffer overflow
 - Terminate program
- If the goal was a Denial-of-Service then it still happens, but the machine is not compromised
- If the canary can be read by an attacker, then a buffer overflow exploit can be made to rewrite the canary

Canaries



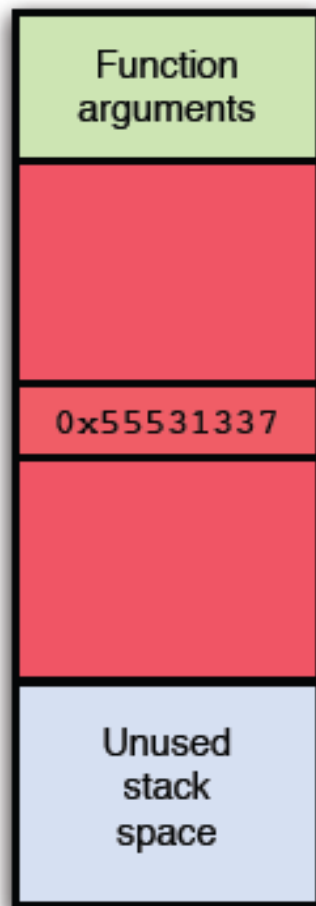
- Technique to detect and prevent buffer overflows by prepending a “canary” to sensitive information
- If canary is “destroyed,” a preceding buffer is assumed to have been overflowed
- Implementations exist for both the stack and heap
 - StackGuard [Cowan97]
 - SSP (aka ProPolice) [Etoh01]
 - dmalloc heap protection
 - Microsoft Visual C++ /GS

Canaries



- Technique to detect and prevent buffer overflows by prepending a “canary” to sensitive information
- If canary is “destroyed,” a preceding buffer is assumed to have been overflowed
- Implementations exist for both the stack and heap
 - StackGuard [Cowan97]
 - SSP (aka ProPolice) [Etoh01]
 - dmalloc heap protection
 - Microsoft Visual C++ /GS

Canaries



- Technique to detect and prevent buffer overflows by prepending a “canary” to sensitive information
- If canary is “destroyed,” a preceding buffer is assumed to have been overflowed
- Implementations exist for both the stack and heap
 - StackGuard [Cowan97]
 - MemGuard
 - SSP (aka ProPolice) [Etoh01]
 - dmalloc heap protection
 - Microsoft Visual C++ /GS

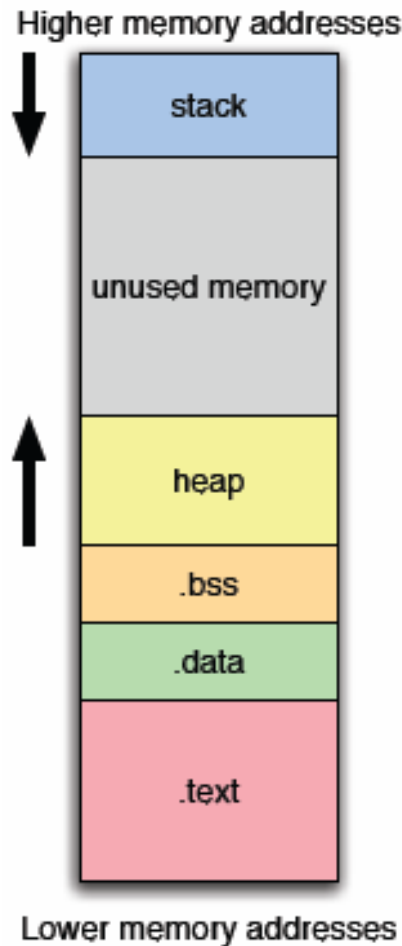
StackGuard

- Compiler extension to gcc
 - prologue pushes random canary on the stack
 - epilogue checks that canary value unchanged
- Assumes return address is unaltered IFF canary word is unaltered
- Can be bypassed if
 - Overflow skips over the canary word
 - Canary word can be guessed
- Only protects against stack smashing attacks

MemGuard

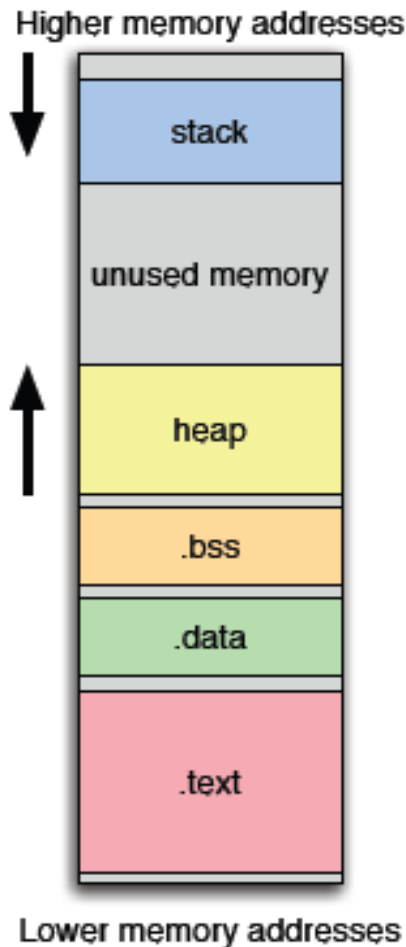
- Protects return address when function is called and unprotects when function returns
- Mark virtual memory pages containing return pointer as read-only and emulates writes to nonprotected words on page
 - 1800 times the cost of normal write
- Use Pentium debug registers to hold return addresses and configure as read only
 - Can only protect top four frames at any time
- Only protects against stack smashing attacks

Address Space Layout Randomization (ASLR)



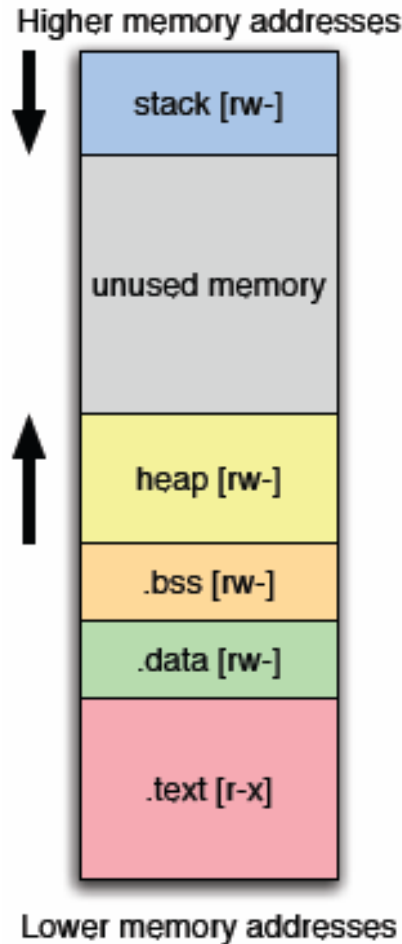
- Technique to randomly perturb locations of memory areas
- Force attacker to guess addresses of important code or data with low probability
- Effectiveness dependent on amount of entropy introduced by scheme
 - increase space within which a memory area may be positioned
 - decreasing the period of perturbation
 - rearranging contents of a memory area
- Various implementations
 - PaX
 - OpenBSD
 - ExecShield

Address Space Layout Randomization (ASLR)



- Technique to randomly perturb locations of memory areas
- Force attacker to guess addresses of important code or data with low probability
- Effectiveness dependent on amount of entropy introduced by scheme
 - increase space within which a memory area may be positioned
 - decreasing the period of perturbation
 - rearranging contents of a memory area
- Various implementations
 - PaX
 - OpenBSD
 - ExecShield

Non-executable memory



- Technique to exclusively allocate memory for either code or data
- May be implemented in hardware as PTE write bit or emulated in software
 - SPARC, Alpha, PowerPC, IA-64 processors
 - PaX
 - ExecShield (RedHat)
 - WX (OpenBSD)
 - NX (AMD processors)
 - XD (Intel processors, identical to NX)
 - data execution prevention, or DEP (recent Windows releases)
- Prevents attacker from injecting data to be executed as code

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
90      nop
90      nop
6a 0b   push   $0xb
58      pop    %eax
99      cld
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Systems that examine events from the network, host, or application for evidence of malicious behavior
- Attacks described by signatures
 - signature can be modeled as a conjunction of constraints on a time-ordered series of events
 - if all constraints for a signature are satisfied, system assumes an attack has occurred, otherwise events are considered normal
- NIDSs contain many signatures for buffer overflow exploits

Misuse-based Intrusion Detection

Shellcode

90	nop	
90	nop	
90	nop	
90	nop	
6a 0b	push	\$0xb
58	pop	%eax
99	cld	

Signature

content:"|90 90 6a 0b ...|"

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
58      pop      %eax
58      pop      %eax
6a 0b   push     $0xb
58      pop      %eax
99      cld
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
58      pop      %eax
58      pop      %eax
6a 0b   push     $0xb
58      pop      %eax
31 d2   xor      %edx,%edx
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
58      pop      %eax
58      pop      %eax
31 c0   xor      %eax,%eax
83 c0 0b add      $0xb,%eax
31 d2   xor      %edx,%edx
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
58      pop      %eax
58      pop      %eax
31 d2   xor      %edx,%edx
31 c0   xor      %eax,%eax
83 c0 0b add      %0xb,%eax
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
58      pop      %eax
58      pop      %eax
31 d2   xor      %edx,%edx
89 d0   mov      %edx,%eax
83 c0 0b add      %0xb,%eax
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Misuse-based Intrusion Detection

Shellcode

```
90      nop
90      nop
58      pop      %eax
58      pop      %eax
31 d2   xor      %edx,%edx
89 d0   mov      %edx,%eax
83 c0 0b add      %0xb,%eax
```

Signature

```
content:"|90 90 6a 0b ...|"
```

- Unfortunately, there are many ways to write shellcode
 - apply semantics-preserving transformations
 - use decoder routine to obfuscate payload
 - use bootstrap routine to fetch different modules
- Matching against specific exploit payloads is fundamentally the wrong approach
- Rather, should attempt to model conditions leading to exploitation of the vulnerability

Moral of the Buffer Overflow Problem

- Always do bounds checking
- Price of bounds checking is efficiency
 - Generally C favors efficiency in most tradeoffs