CS3235 - Computer Security Tenth lecture - Machine architecture 1

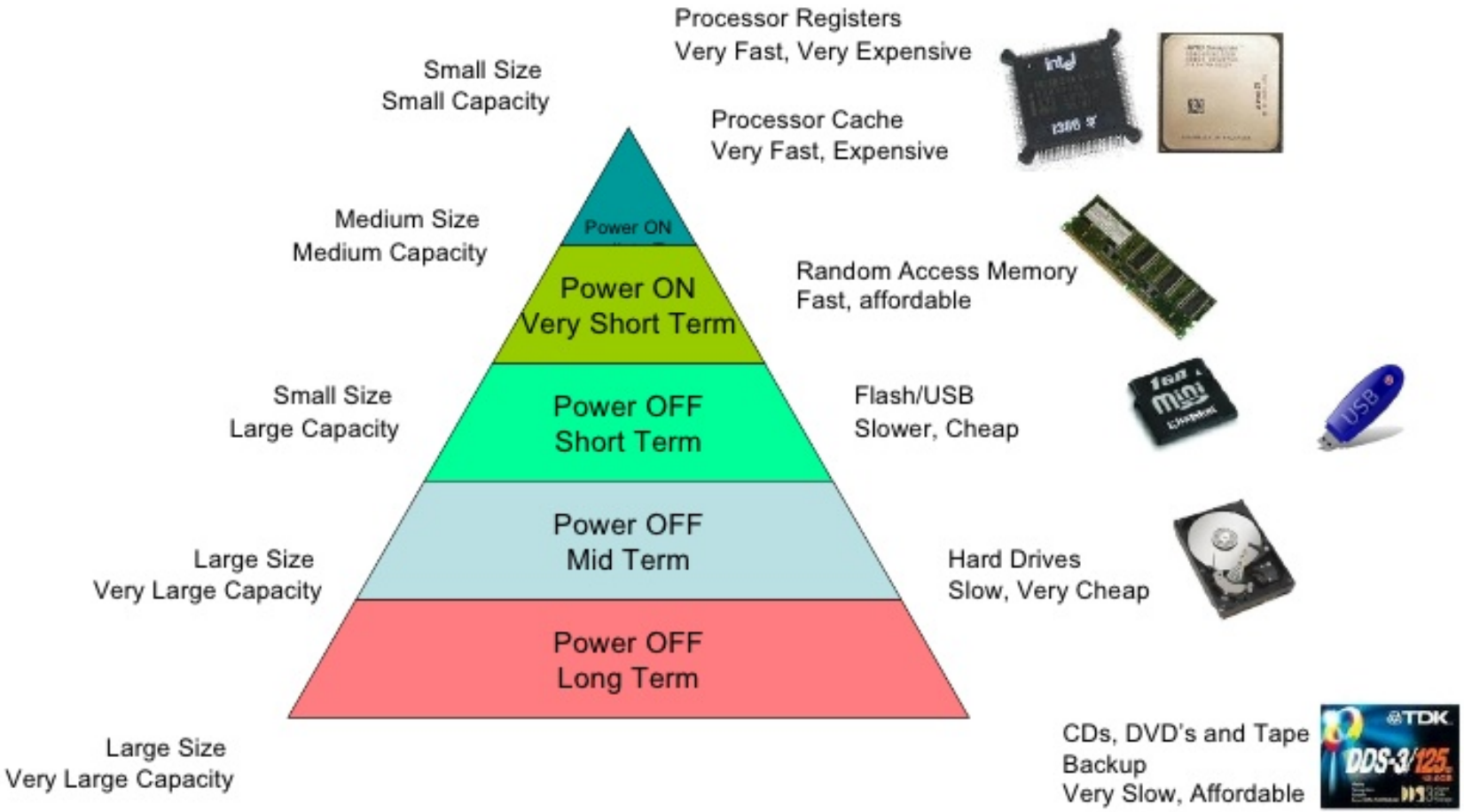
Hugh Anderson

National University of Singapore
School of Computing

November, 2018

Memory...

Computer Memory Hierarchy



Outline

Outline

How programs use memory for data

Storing a C string will come back to haunt us...

In C, a string like `Hello` is stored in six successive bytes, the five characters, and a NULL terminating character. This allows compact representation, and fast access to a string.

In Java for contrast, a string is stored as an object. For a five character string, the storage takes up 48 bytes: An object containing the actual characters; an integer offset into the array at which the string starts; the length of the string; and another int. This results in non-compact representation, and slow access.

When memory is needed for a program, the OS allocates memory:

- The calls to access memory are `malloc()` and `free()`.
- Java maintains its own heap, separate from the system heap.
- When lots of allocations and deallocations are being made, the heap may become fragmented, and the OS will run a garbage collector to fix up the heap from time to time.

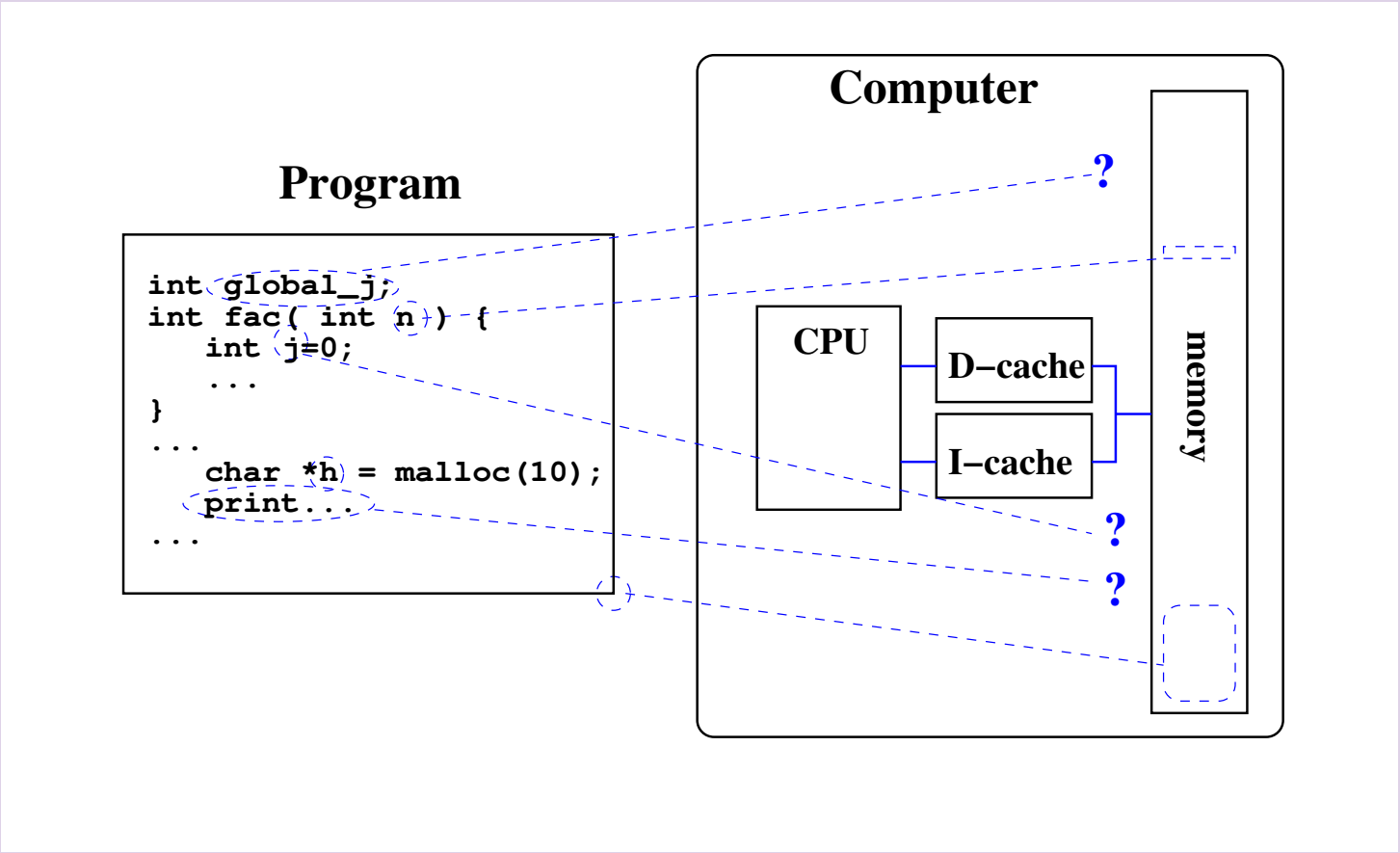
Machine architecture: visualizing memory

Location of “variables”?

```
int global_j;
int fac( int n ) {
   int j=0;
   if n=0 then
      return 1
   else
      return n*fac(n-1)
}
...
   char *h=malloc(10);
   global_j=5;
   print(fac(global_j));
...
```

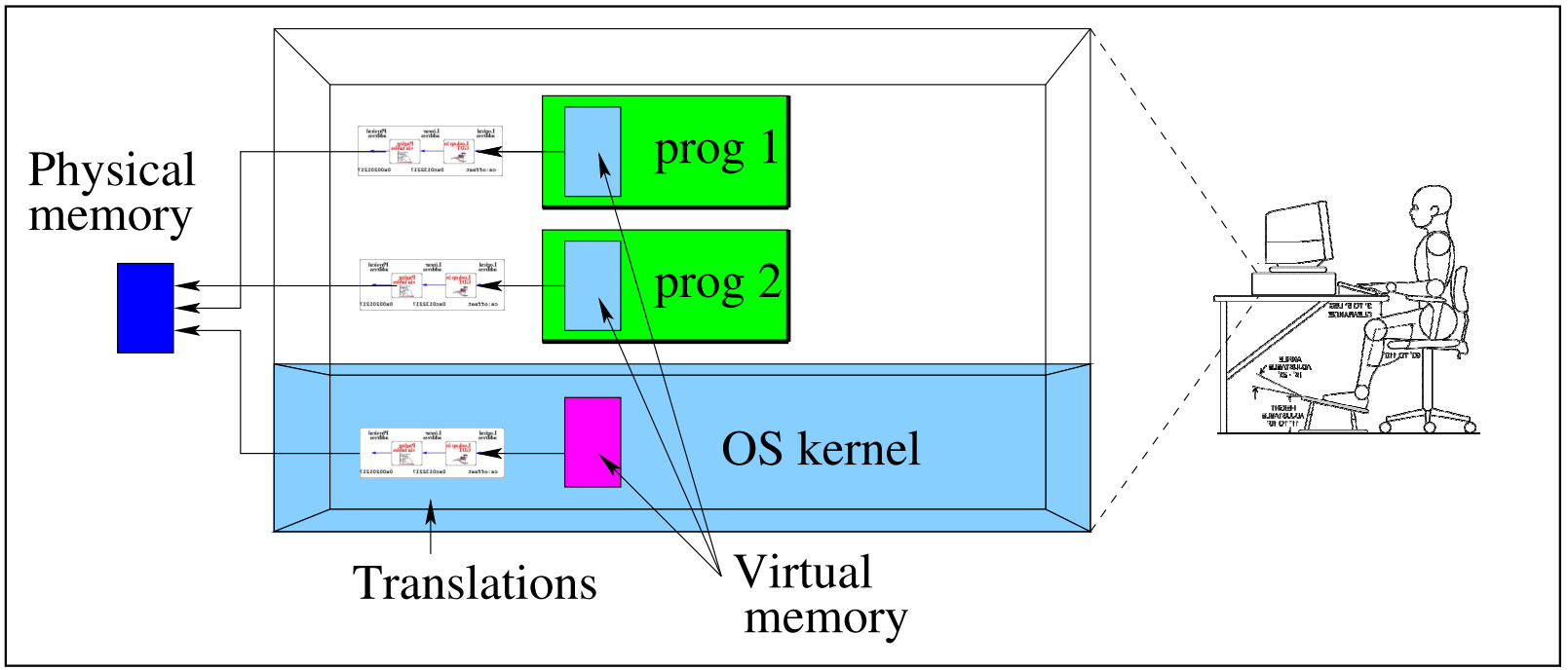
Machine architecture: visualizing memory

Location of “variables”?



Process use of memory

Virtual memory, physical memory, and translations...



The processes, and the OS kernel, each have their own view of memory, somehow mapped/translated to the real (physical) memory.

Virtual memory addressing

Overview

Modern operating systems and hardware give each process an individual address space - starting at **0x0**.

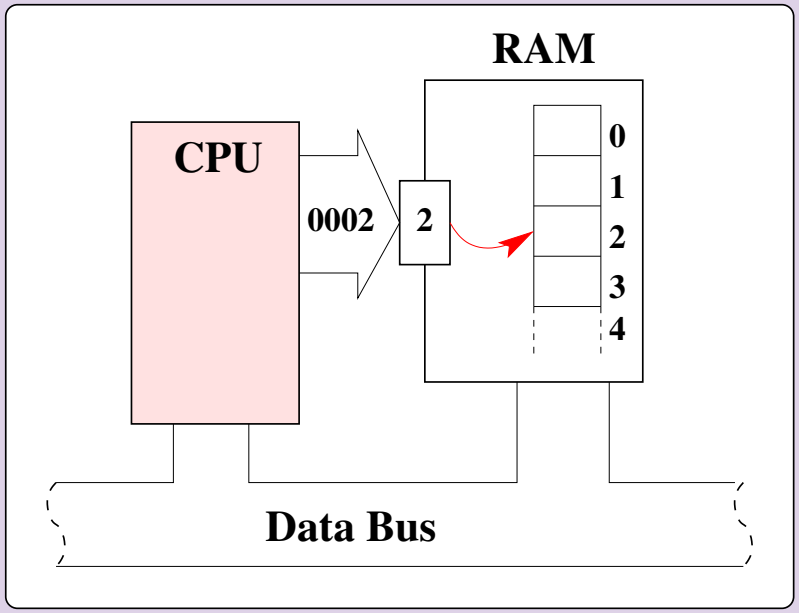
A translation unit maps this *virtual* address to a real *physical* address, in fixed size pages.

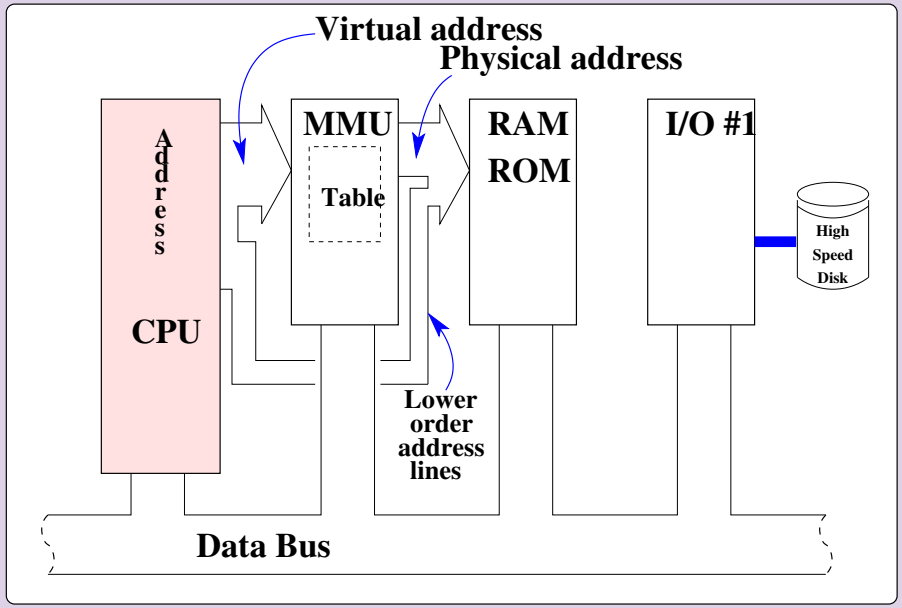
From the world of Intel...

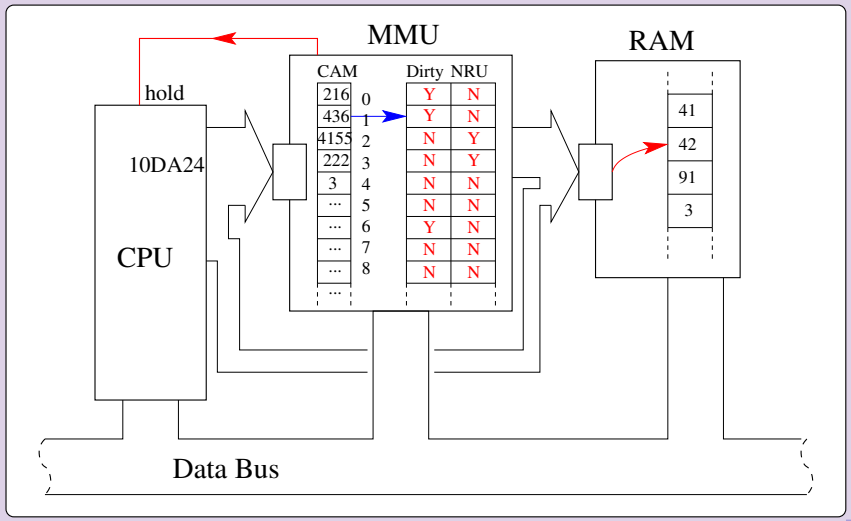
On the Intel chips, there are 32/64 address lines. For a 32-bit system each process could have a virtual address space of up to 2^{32} bytes - or 4 gigabytes.

In a paged system, if our page size was 4096 bytes, the address bus would be split, with the bottom 12 bits going directly to the memory and the upper address bits to the translation unit.

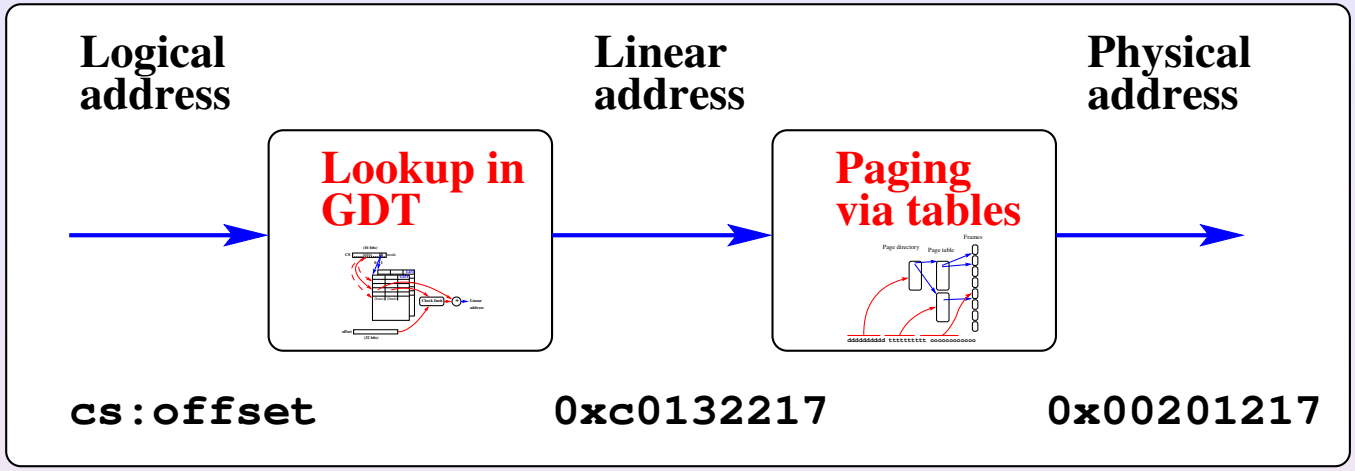
Simple (embedded systems) and VM







VM=paging+segments (+disk)...



Disk can be used to store contents of memory

The part of the disk used in this way is called the ***'swap'*** - it may be either a file used for the purpose, or a partition of disk.

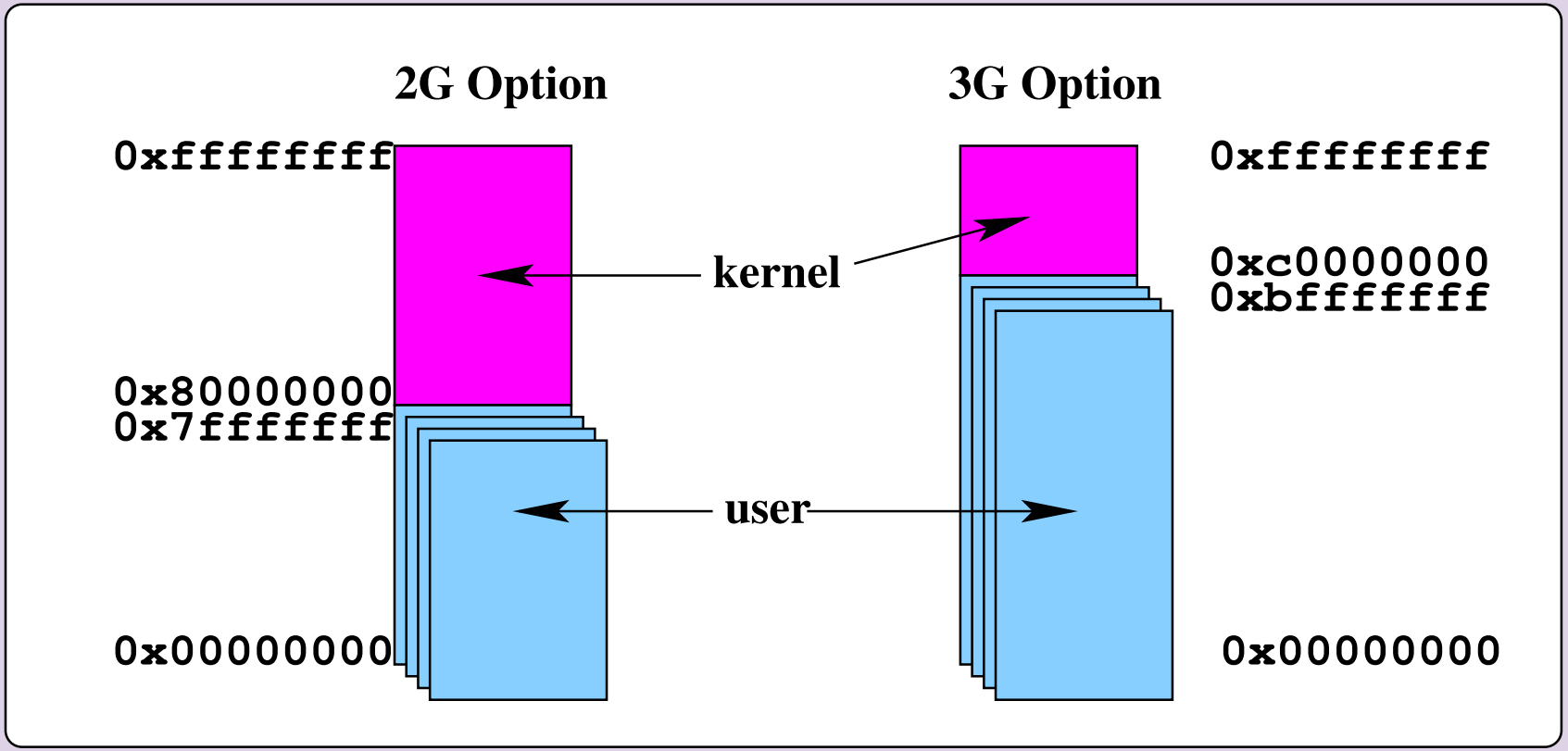
Page fault

When a VM system attempts to find a page, and does not find one, it is known as a ***page fault***.

When a page fault occurs, the OS must *replace* a page in memory, efficiently, with the new one.

Windows (32-bit) memory allocation

Several space options

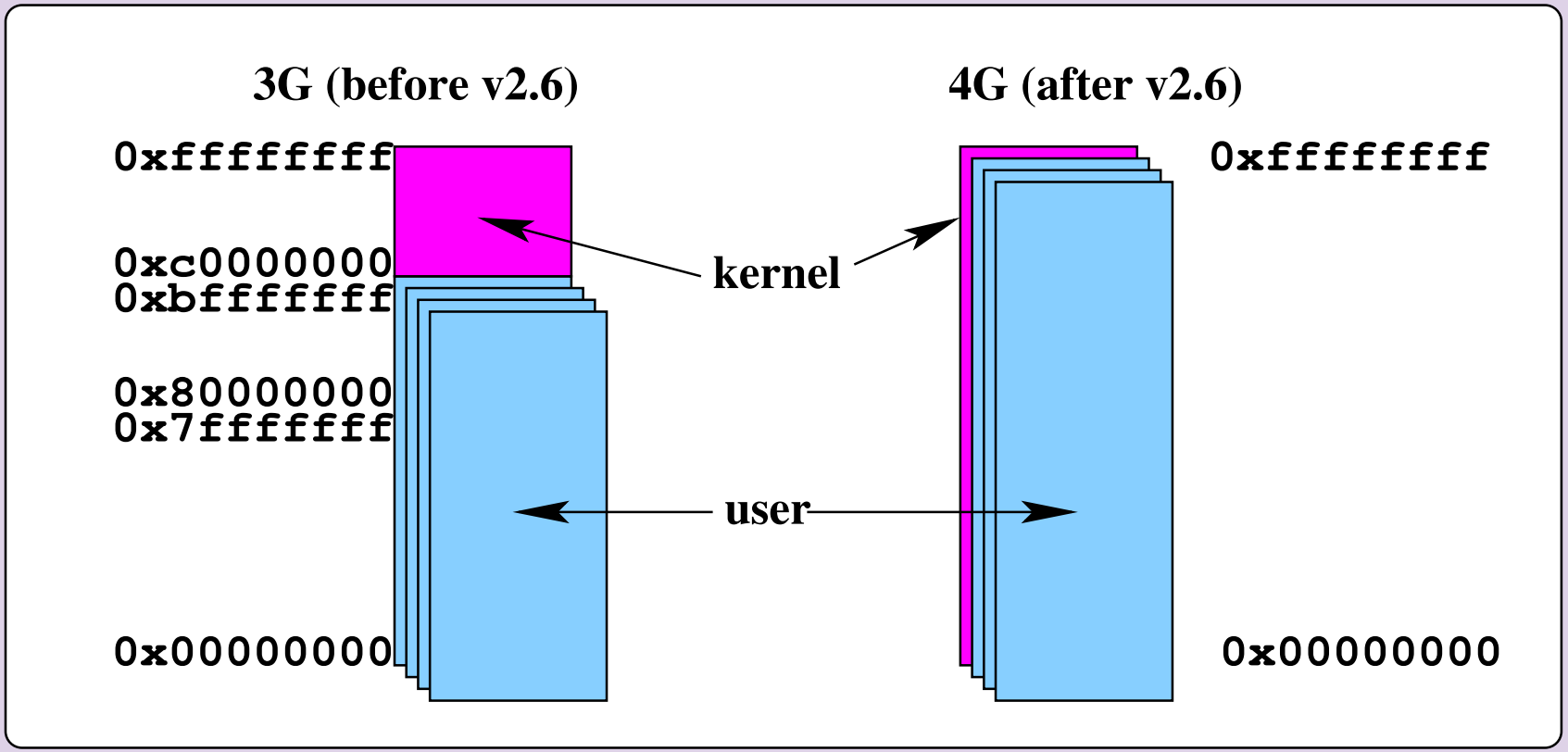


The “Kernel” area of memory has the HAL, drivers, page tables...

The “User” area contains your exe code, dlls, stacks...

Linux (32-bit) memory allocation

3G before kernel v2.6, 4G afterwards

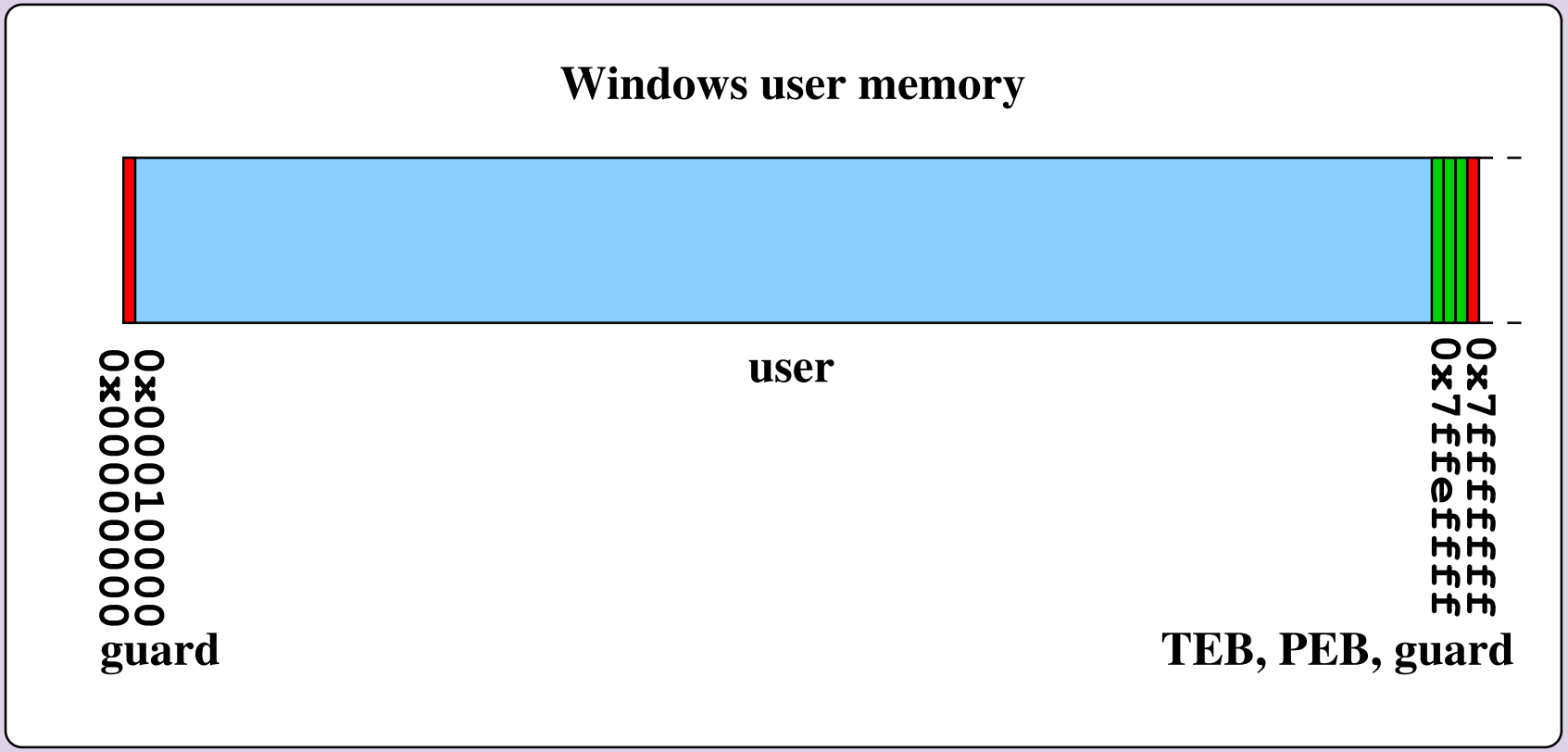


The “Kernel” area of memory has drivers, system stacks, page tables...

The “User” area contains executable code, libraries, stacks...

Windows process memory allocation

...for 2G model...

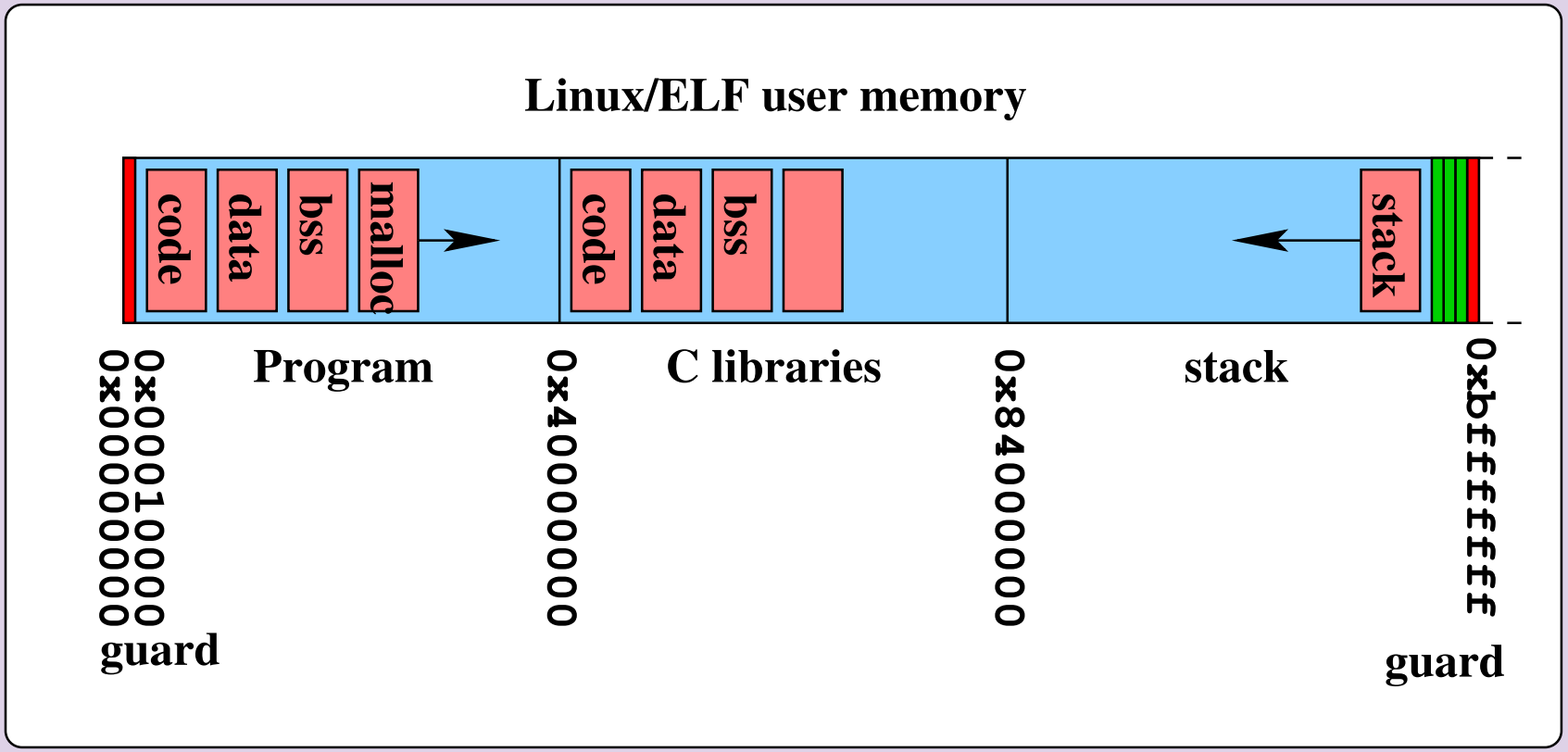


TEB and PEB are thread and process environment blocks.

The 64K guard blocks are sacrificial... against bad pointer references.

Linux ELF format user segments

Program, shared libraries, stack...



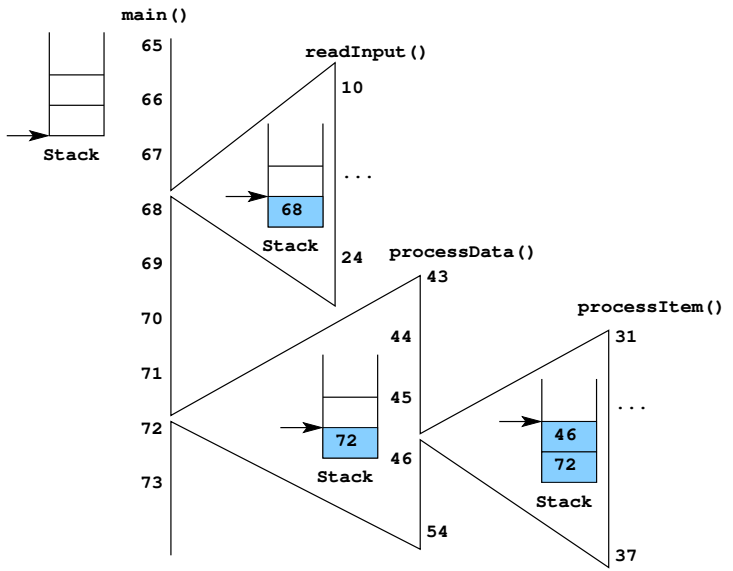
The malloc'ed memory is "above" the program.

As always, the stack grows downwards.

How programs use memory for return addresses

Return addresses need to be nested, so we use a stack:

```
10 readInput() {
..    ...
..    ...
24 }                        // Return to the most recent saved away address
...
...
31 processItem() {
32    ...
..    ...
37 }                        // Return to the most recent saved away address
...
...
43 processData() {
44    ...
45    processItem();        // Save away 46 and call processItem()
46    ...
..    ...
54 }                        // Return to the most recent saved away address
...
...
65 main() {
66    ...
67    readInput();          // Save away 68 and call readInput()
68    ...
69    ...
70    ...
71    processData();        // Save away 72 and call processData()
72    ...
73 }
```



A little warning...

Later, we will use stacks that are upside down, because that is how computers use stacks - they grow towards lower/smaller addresses.

Memory attack 1: simple Program

A pretty simple program...

CODE LISTING **vulnerable.c**

```
void
main (int argc, char *argv[])
{
    char buffer[512];

    printf ("Argument is %s\n", argv[1]);
    strcpy (buffer, argv[1]);
}
```

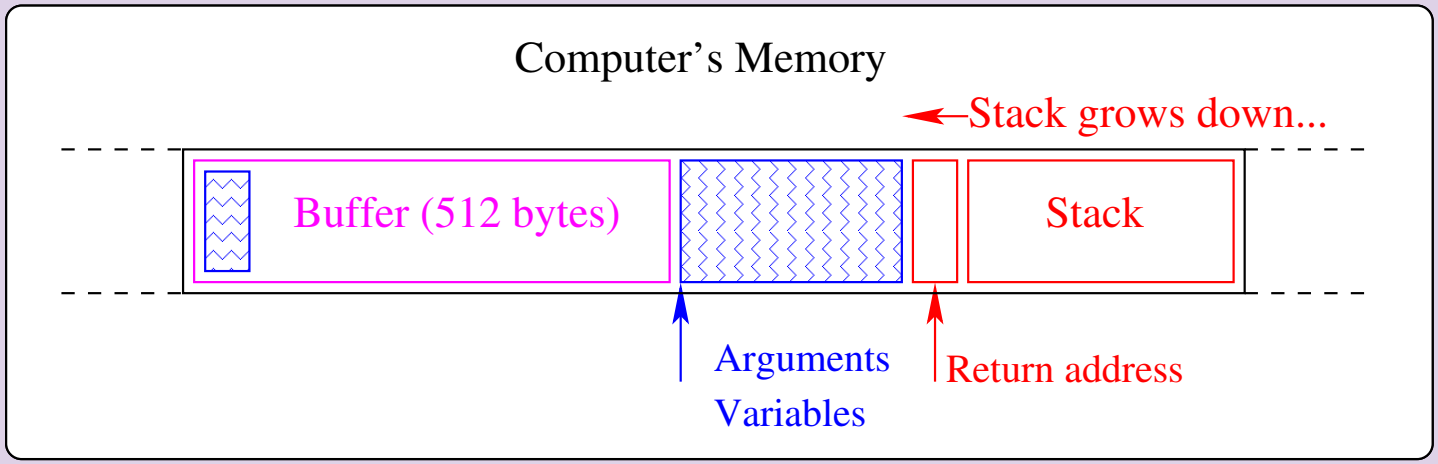
When we run it:

```
[hugh@pnp176-44 programs]$ ./vulnerable test
Argument is test
[hugh@pnp176-44 programs]$
```

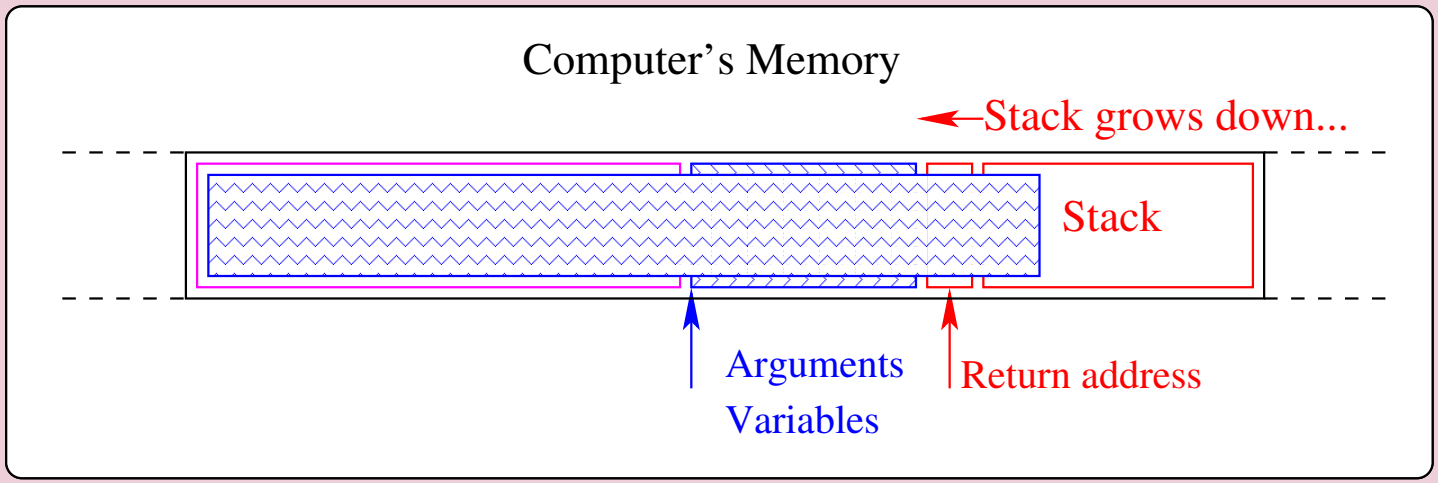
but...

Memory attack 1: stack buffer overflow

Normal operation

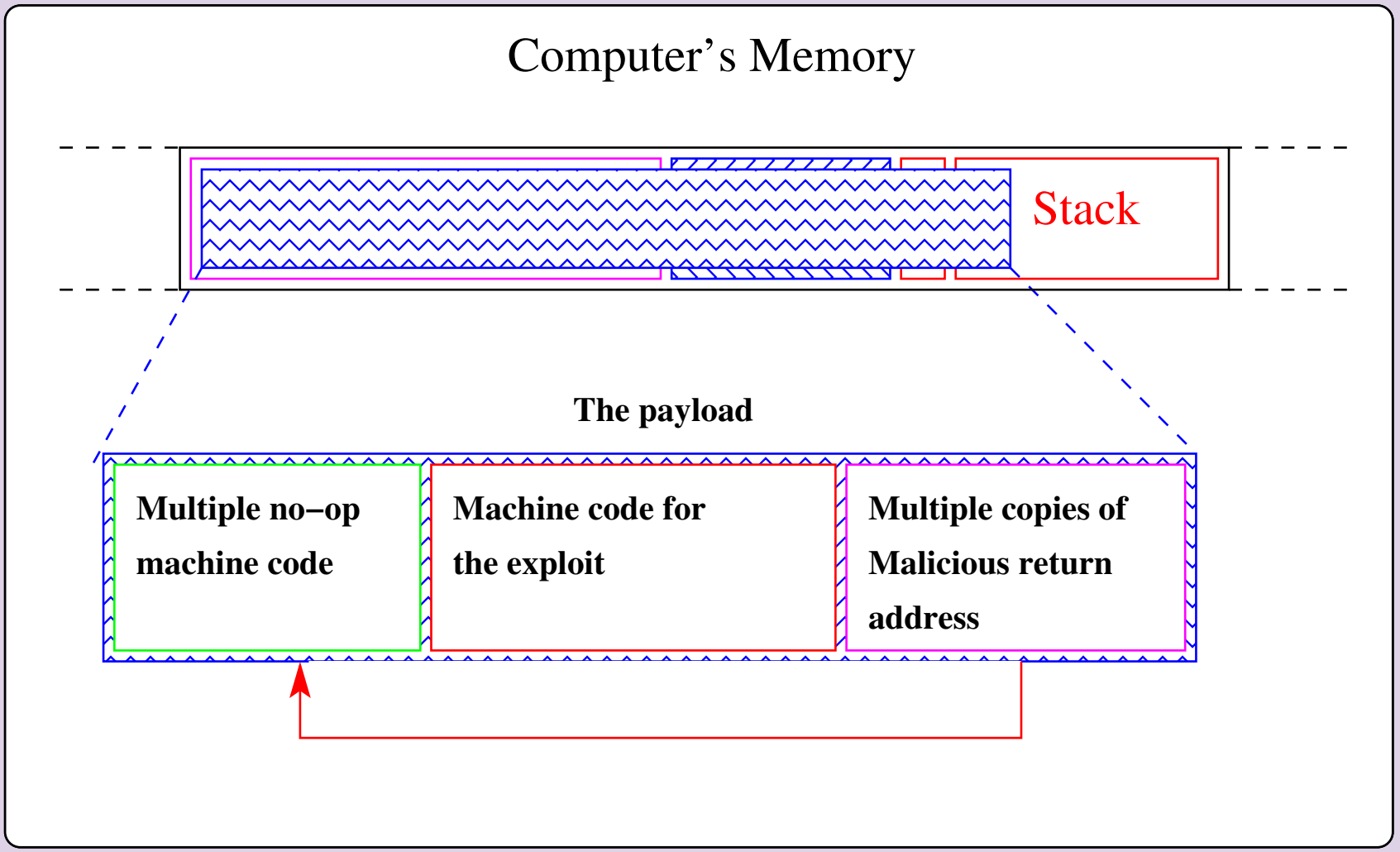


Overwrite the end of an array, with an "EGG"



Smashing the stack - the payload!

Contents of the payload



Payload code for an exploit

A C program which calls a shell...

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

```
movl    string,string_addr
movb    $0x0,null_byte
movl    $0x0,null_addr
movl    $0xb,%eax
movl    string,%ebx
leal    string,%ecx
leal    null_string,%edx
int     $0x80           // OS call
"/bin/sh" string goes here.
```

At the left we see some C source code for running the program /bin/sh. On the right we see assembler code with extra OS nonsense removed.

Note that the binary code program has "zeroes" in it, and these will have to be removed if strcpy is to copy the program onto the stack. We can use translations like:

```
movb    $0x0,null_byte          xorl    %eax,%eax
                                movb    eax, null_byte
```

Using the buffer overflow attack

3 examples of situations in which we can use it

1. A server (say a web server) that expects a query, and returns a response. The demo buffer overflow attack done in class is one of these.
2. A CGI/ASP or perl script inside a web server
3. A SUID root program on a UNIX system

Find a program that has a buffer, ... and that does not check its bounds. Then deliver an EGG to it to overflow the buffer.

Overwrite stack return address with address of OUR code.

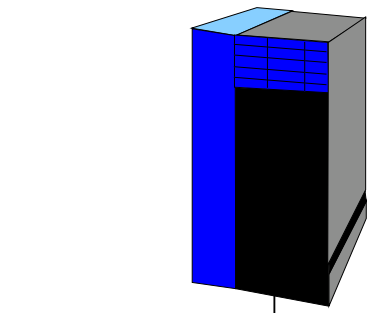
The program then runs OUR code.

Example attack

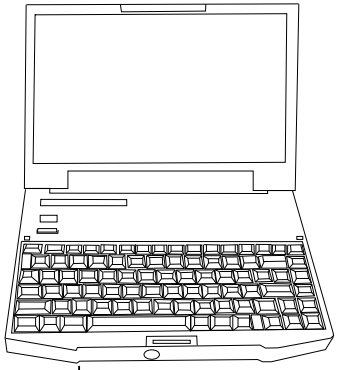
Many attacks on Microsoft systems are based on various buffer overflow problems. CA-2003-20 W32/Blaster worm:
The W32/Blaster worm exploits a vulnerability in Microsoft's DCOM RPC interface as described in VU#568148 and CA-2003-16. Upon successful execution....

Memory attack 1: example

Consider the following scenario...



VM Honeypot on Hugh's mac



Hugh's mac

(Virtual) web server

(hack from here)

The Virtual machine running on the mac is running GNU+Linux, and a (cut down) version of an old version of Apache web server

Memory attack 1: web server code...

And in the web server we have this code:

```
void process(int newsockfd) {
   char line[512];
   ...
   ...NEXT BIT DOESNT CHECK ARRAY SIZE! i.e. n<511
   while (n>0 && c!='\n') {
      n = read (newsockfd, &c, 1);
      ... add it to line[idx++]...
   }
    ...
   return;
}
```

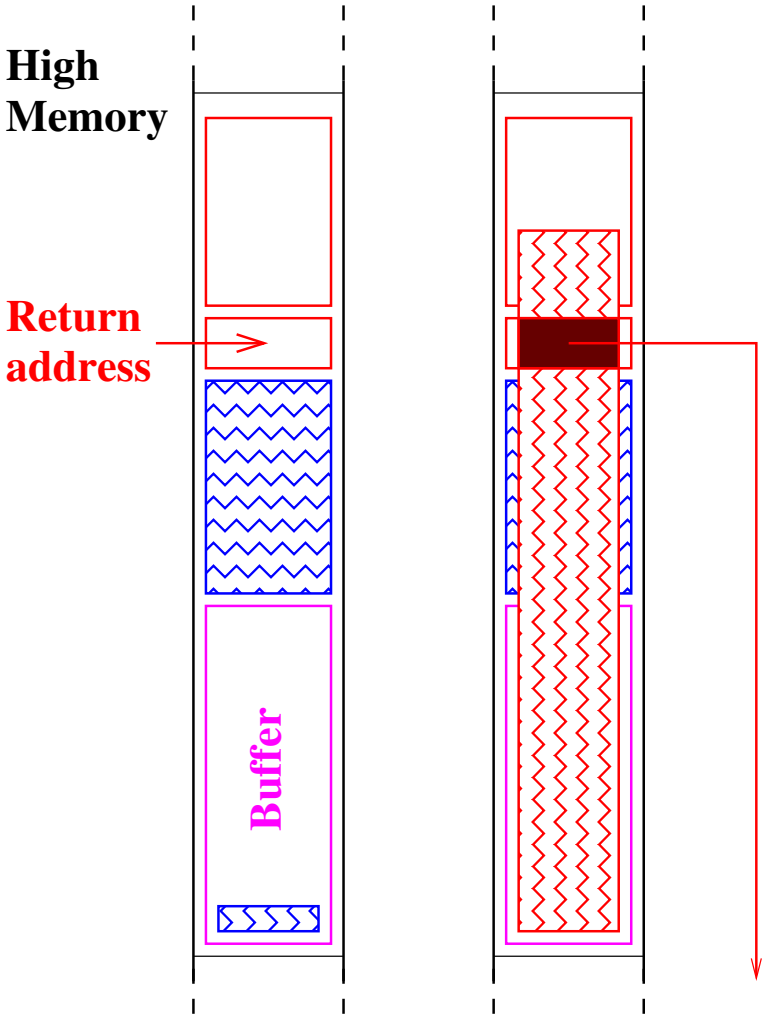
The general operation of this web server is hackable...

It matches our attack mechanism: A web server receives a file spec (index.html), and does not properly check a buffer that it puts the spec into.

We replace the file spec with EGG. Note that we will have to use a specialized program to deliver the EGG (i.e. not firefox)

Memory attack 2: return to libc!

Run code already in libc...

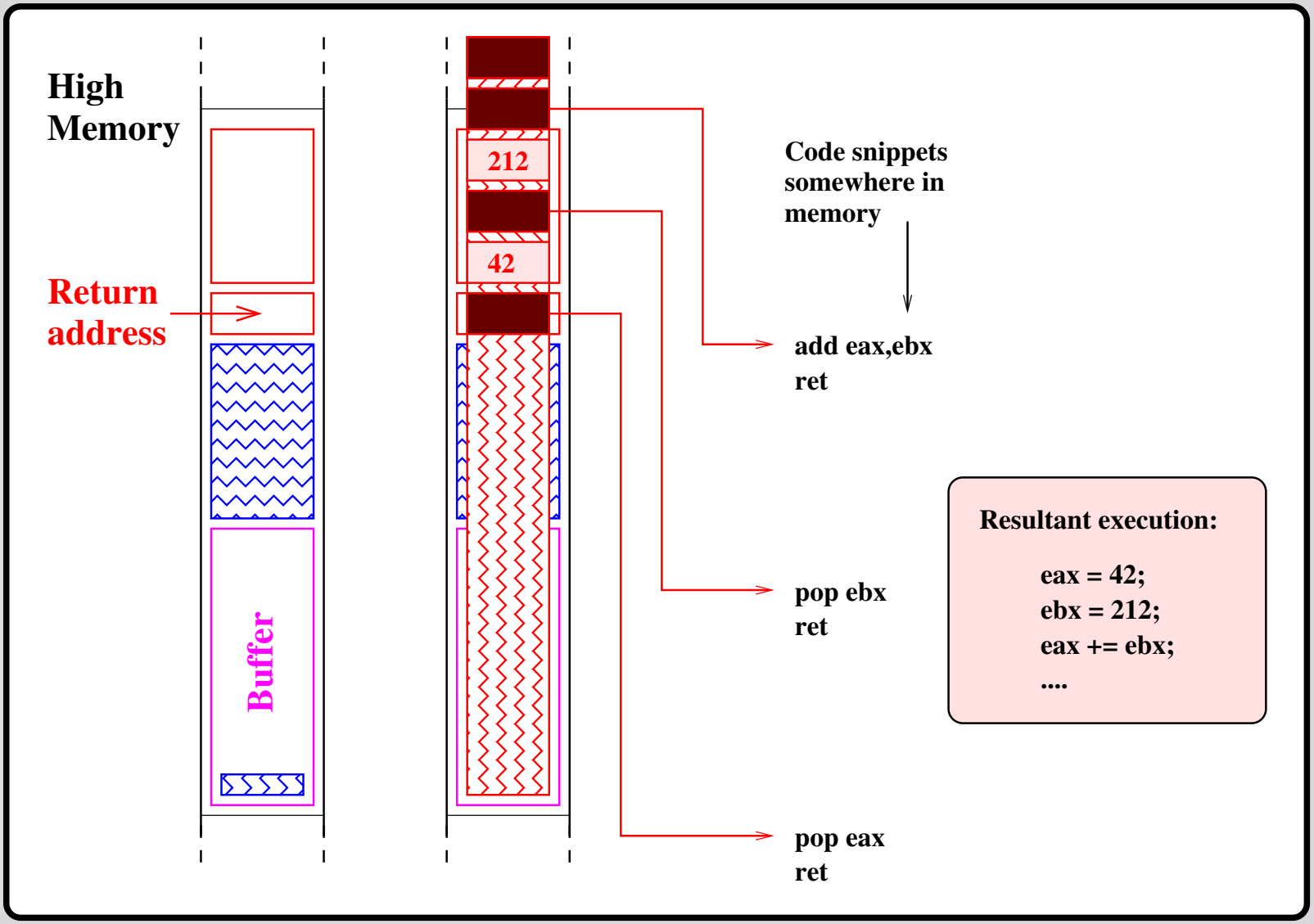


This has the address of
the libc system call system().

You still need to provide an
argument to the system call...
so you get system("/bin/sh")

Memory attack 3: return oriented programming

Even with DEP: find code snippets, and overwrite stack...



Memory attack 4: heap overflow

The younger sibling of the buffer overflow

The heap is managed by the OS as a doubly linked data structure, containing a header, and the memory to be allocated.

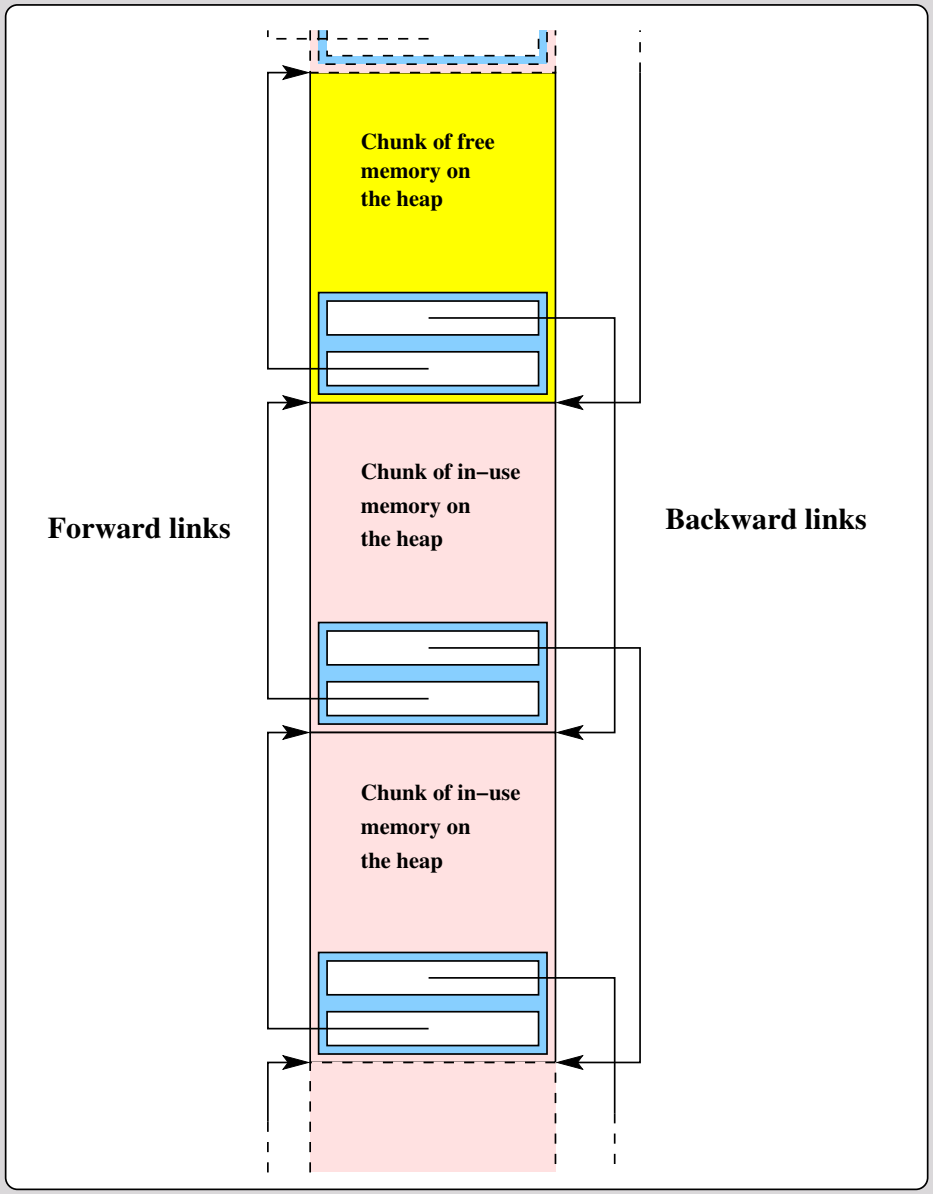
The attacker starts by constructing/coding something that results in the OS allocating and deallocating memory from the heap. Then:

- After the `malloc()`, the program writes to the memory, and then returns it using `free()`. By careful choice of what we write to the memory, we can overwrite the NEXT header in the heap.
- The OS then merges this *freed-up* memory, manipulating the headers in order to do so. It rewrites values in the previous headers that are dependent on the values in the next headers.
- If the next header is modified to point to some place where a return address is being kept, then we can run crafted code.

Background reading:
http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html

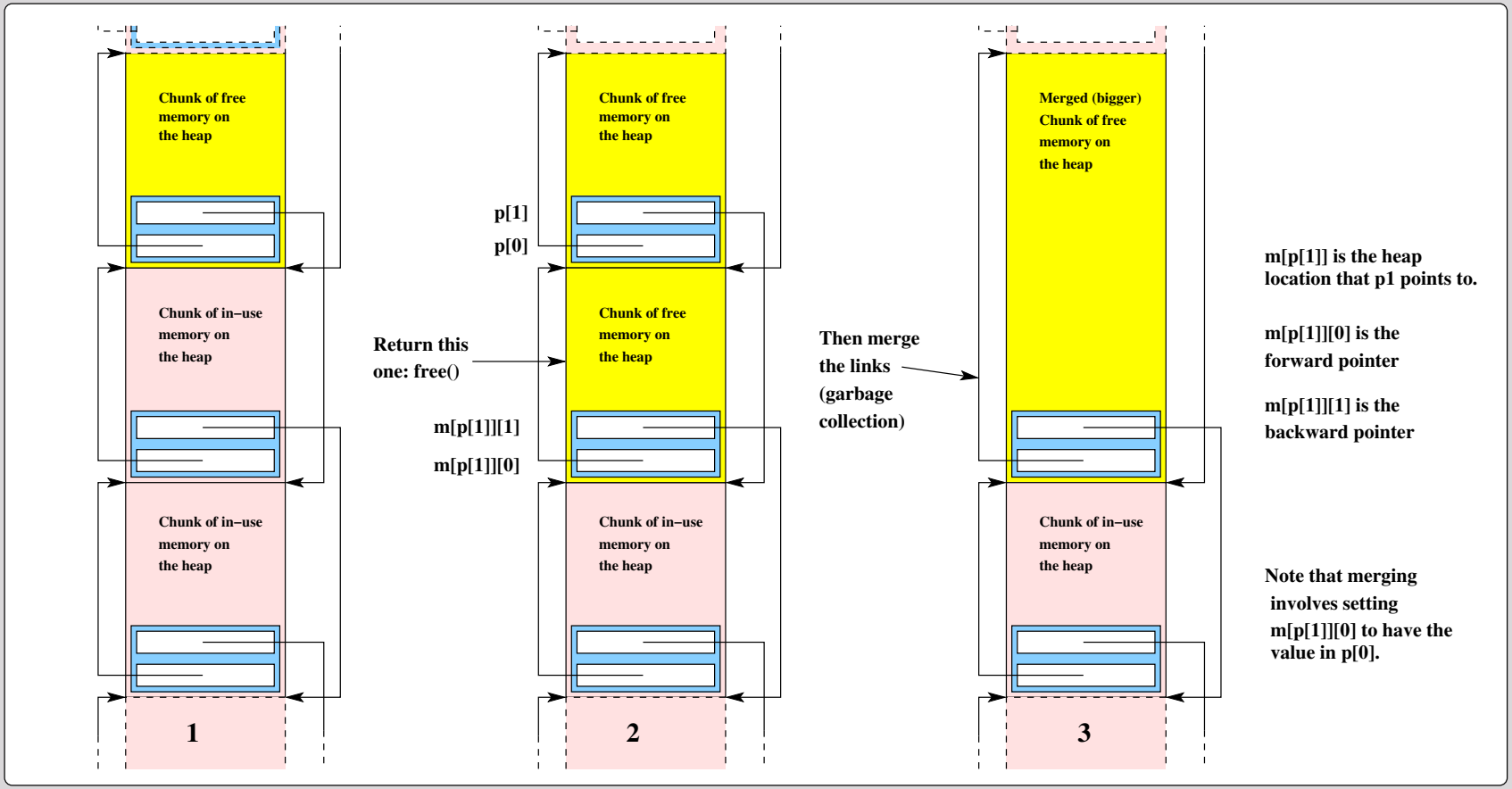
Memory attack 4: heap overflow

Consider the following part of a heap...



Memory attack 4: heap overflow

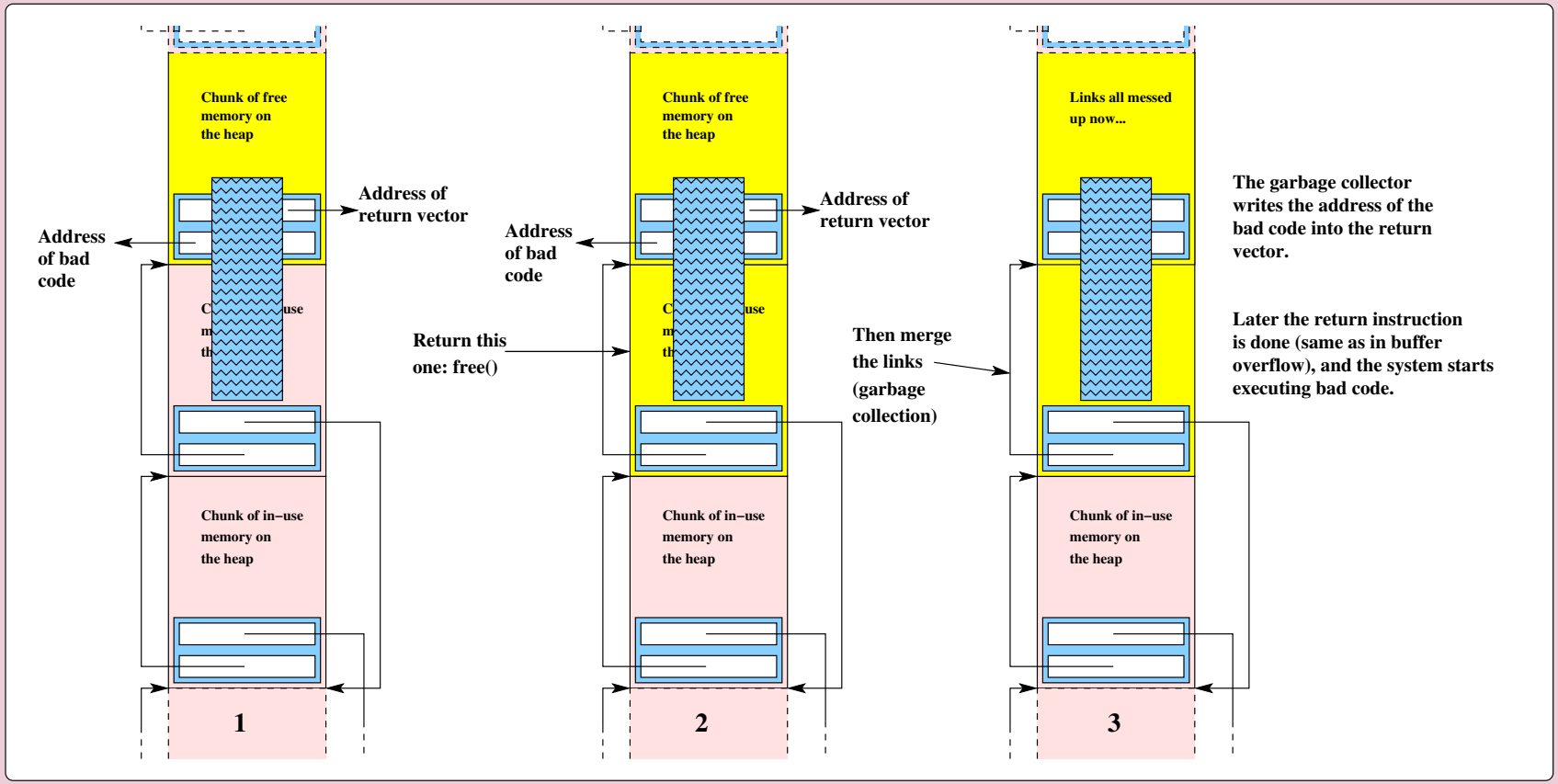
The normal free() and merge...



The OS heap management software automatically rewrites the values of the forward and backward links, using the values in the *next* links.

Memory attack 4: heap overflow

Consider the scenario where attacker changes pointers...



The attacker uses the OS GC to write the address of malicious code into a *return-address* location, as for the buffer overflow.

Stack protection

Three methods to make overflow attacks harder:

1. Not allow execution of code in the stack segment (as in MacOSX/openBSD) - but ROP still can get through.
2. Randomly move the starting stack for processes.
3. Put a value just below the return address (a canary), and check it before doing a return from subroutine. It is hard to overwrite the return address without overwriting the canary.

Windows 7 includes a wide range of protection against stack/heap overflows: They include the removal of commonly targeted data structures, heap entry metadata randomization, the heap header is more complex, there are randomized heap base addresses, function pointer are encoded, as well as the normal DEP and ASLR.

Memory attack 5: OpenSSL Heartbeat

What is a heartbeat in openssl?

A heartbeat is a recent (February 2012) extension to TLS, to allow a client to check if a TLS server is still alive (rather than tearing down a connection and renegotiating).
It is described in https://tools.ietf.org/html/rfc6520.

The general idea is that clients can request a heartbeat, sending a heartbeat_request message with this structure:

```
struct {
      HeartbeatMessageType type;
      uint16 payload_length;
      opaque payload[payload_length];
      ...
} HeartbeatMessage;
```

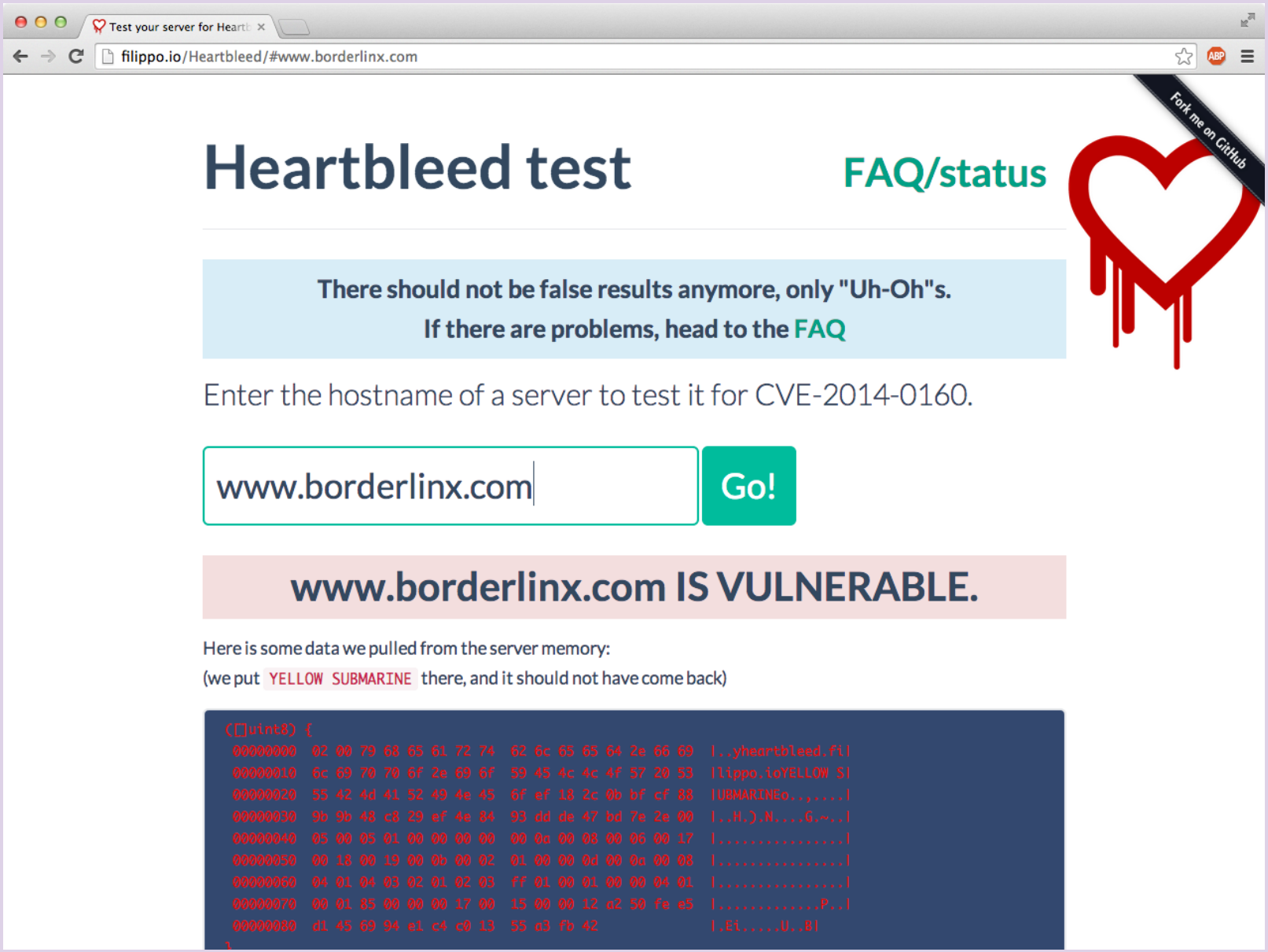
The server will send back the payload in a matching heartbeat_response.

What is the issue?

What if the client lied about the payload_length?

Memory attack 5: OpenSSL Heartbeat

Request and response different...



Memory attack 5: OpenSSL Heartbeat

Original openssl-1.0.1f/ssl/d1_both.c...

```
/* Read type and payload length first */
     hbtype = *p++;
     n2s(p, payload);
     pl = p; ...
```

The original source gets the type from the record that p points to, and puts it into hbtype. It then copies the next two bytes into the variable payload. This is the length of the payload, and is done without checking. The variable pl is the actual payload, which is later echoed, using the length value without any more checking.

Fixed openssl-1.0.1g/ssl/d1_both.c...

```
/* Read type and payload length first */
     hbtype = *p++;
     n2s(p, payload);
     if (1 + 2 + payload + 16 > s->s3->rrec.length)
               return 0; /* silently discard */
     pl = p; ...
```

(There is a little more, but that is the general idea).

Memory attack 6: Memory after reset

After a hardware reset...

If you press the reset button on the computer...
the memory of the computer retains the same values it had just before the reset.

If you are careful, it is possible to

- bypass the memory test (which destroys some values in memory).
- boot a small OS (such as DOS) which will not change any of the important memory contents.
- use a DOS application to extract the raw contents of memory, rebuilding the paging and segment tables, and then
- re-create the memory for each process.

The memory may contain (for example) secrets that had been left in memory.

Memory attack 7: Malloc hack

The scenario...

In 1999, Microsoft fixed a security error in the login "password" software in the Windows login.

The error (it is not a "bug") went something like this:

- The software allocated space in memory for the user to type in the password using **`malloc()`**.
- Then the user typed in the password (and it ended up in the memory of the computer), and it was used and then
- the memory used was returned to the OS using **`free()`**:

```
// ... Allocate some memory for the password:
buffer = malloc( MaxPasswordSize );
// ... Put the password into this buffer,
// ... and use it to log in.
// ...
// ... Then return the memory to the OS:
free( buffer );
```

Memory attack 7: Malloc hack

The error...

The problem with this scenario, was that the password was left in clear text in the memory of the computer, and the NEXT user of the computer could search the heap of the computer and discover it.

The fix was to clear the buffer using bzero():

```
// ... Zero out all the contents of the buffer:
bzero( buffer, MaxPasswordSize );
// ... Then immediately return the memory to the OS:
free( buffer );
```

The error returns...

In 2001 hackers discovered that (after *improvements* made to the Microsoft compilers) the password was back in clear text in memory!

What can possibly be wrong with this *obviously correct* code?

Memory attack 7: Malloc hack

Error returns to haunt Microsoft...

The problem lay with the compiler:

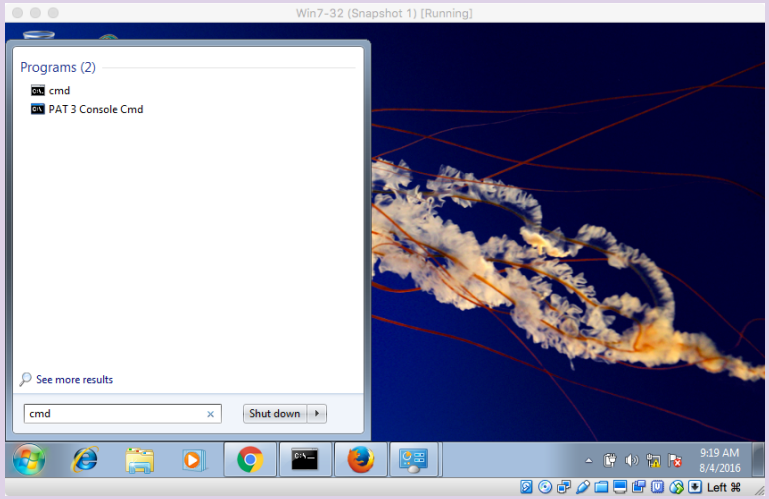
- The newer compiler did a lot of optimization, and
- recognized that the buffer was not used again after setting all its values to zero, so
- the `bzero()` was (silently) removed!

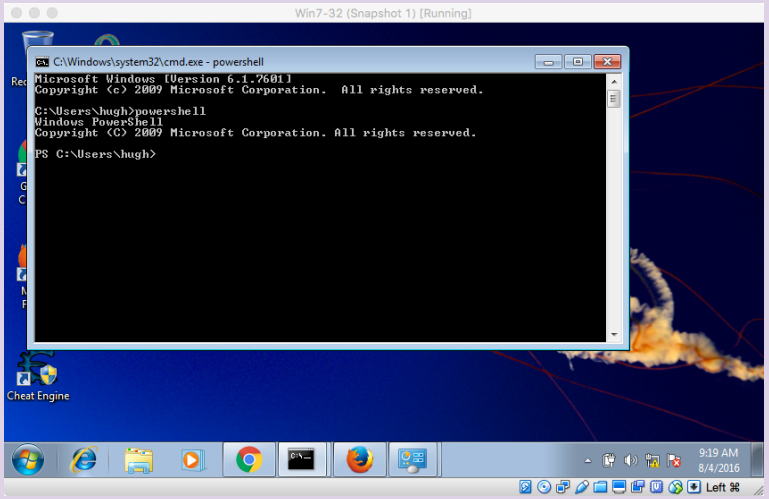
Score 10 for the compiler, and simultaneously, 0 for the compiler.

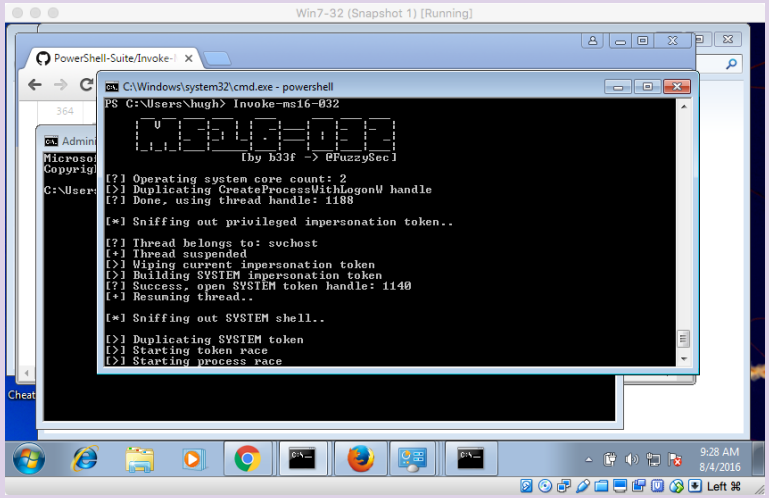
Surface 7 attack, privilege escalation...

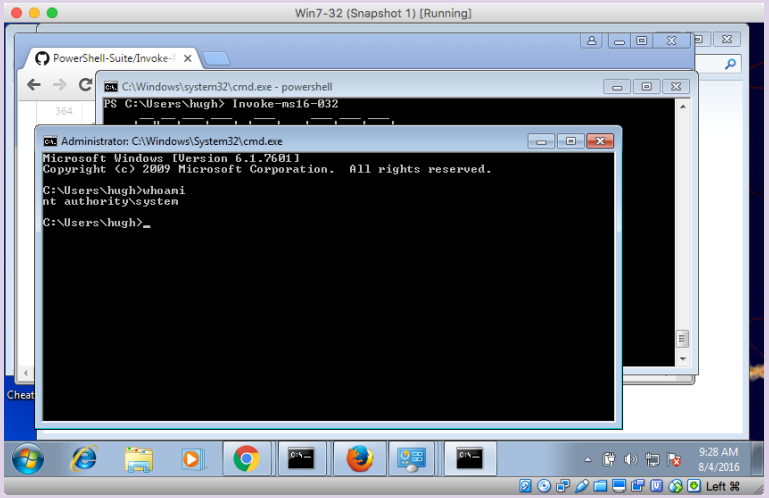
A recipe...

Login to Windows and run powershell. Run `Invoke-ms16-032.ps1`...









Outline

Some useful practices

For system developers or coders...

- Check for use of safe memory technologies (in OS, compiler, libraries...)
- Check the use of parameters and buffers/arrays
- Overwrite secrets as soon as possible (and watch for compiler optimization)
- White hat approach: code conservatively -
  - Make interfaces explicit rather than implicit.
  - Minimize interfaces.
  - Check and recheck safety properties.
  - Requires, ensures and invariants.
- Wear a black hat

Safe memory technologies

As a system designer or coder...

You should be aware of the environment within which you are working.

For example:

- Your compiler - does it implement some sort of stack protection (such as StackGuard). Is there a better system available?
- The library routines you use (especially in Microsoft). Do they implement good stack protection technology?
- Your compiler - what sort of optimizations does it do?
- Your runtime system - does it use the OS heap or it's own? What known attacks are there with this runtime system/heap?
- Your OS - is it recent? Are there any known attacks on it?
- Your language - does it have many security weaknesses? Is it C, C++, C#?
- Your code - have you used safer library routines? For example `strncpy()` versus `strcpy()`.

Parameters and buffers/arrays

Consider your source code. Ask yourself..

- Is there any way at all that your program takes input from outside (In 99% of programs the answer is YES BTW).
- Do you ensure that the input is constrained to be within the range of values you expect? As early as possible?

Can an index can be outside the range you have specified?

```
idx=10;
while (idx>0) {
    ... body ...
}
```

Consider the above code - does the body of the while loop get executed at least once? (i.e. Ensure no *indirect* interfaces between code segments.)

There is no way of avoiding secrets in memory...

...but once you have used a secret, you should immediately overwrite it. This will reduce a window of opportunity to an attacker.

Conservative code: modularity

Software design, modular programming...

Modular design is an effective way of designing, building, and coding large software projects:

http://en.wikipedia.org/wiki/Modular_programming

In a good modular design concerns are separated such that

- modules are more or less stand alone, and that
- when they do depend upon other modules of the system the interfaces are explicit (i.e. easily visible in the code).

The system designer/coder should reduce or minimize the dependencies.

Conservative code: explicit interfaces

Do not hide away interaction between modules...

If two modules share a variable, this should be explicit, and not hidden:

```
var x;
module1() {
   manipulate x...
}
...
module42() {
   manipulate x...
}
```

In the above code - two modules both rely on the shared variable, but if you read the code for `module1()`, there is no clear indication that it is also being manipulated by module42(). When you reason to yourself about the correctness of `module1()`, you may forget the effect of `module42()`.

It is possible to reason about *tiny* segments of code in a program, but ...
... even the fanciest designer or coder cannot correctly reason about *small* programs, let alone *large* ones.

Conservative code: interfaces

If variables are being used across modules...

Make their use explicit:

- ... as parameters (even though you may believe this to be inefficient),
- ... by using modules (classes/objects) correctly,
- ... by documentation and warnings.
- ... revisit your code and review it for good programming practices.

Minimize them ... make sure you actually do need them, and if not, remove them.

Conservative code: design by contract

Software design, design-by-contract, code-contracts...

An approach to defensive programming.
Design-by-contract is an effective way of checking and rechecking safety properties:

http://en.wikipedia.org/wiki/Design_by_contract

In a good design-by-contract design, the components and interfaces between them are precisely defined using

- preconditions (requires). What does the component expect?
- postconditions (ensures). What does the component guarantee?
- invariants. What does the component maintain?

These definitions are called *contracts*. Bertrand Meyer coined the design-by-contract term when developing the language Eiffel.
There is at least some support for contracts in some languages, including Java, Spec#, .NET.

Conservative code: requires, ensures

A useful idea

In Eiffel, `requires` and `ensures` are part of the language and are checked at the beginning (for `requires`) and the end (for `ensures`):

```
my_list.do_all (agent (s: STRING)
     require
          not_void: s /= Void
     do
          s.append_character (',')
     ensure
          appended: s.count = old s.count + 1
     end)
```

In the above code, if either contract is broken, the run time system traps to some handler for the error.

If there is no handler, the program exits, with the name of the broken part of the contract.

Conservative code: invariants

When you want to ensure a safety property is always kept

```
class interface
    ACCOUNT
feature -- Access
      balance: INTEGER              -- Current balance
      deposit_count: INTEGER        -- Number of deposits
feature -- Element change
      deposit (sum: INTEGER)        -- Add 'sum' to account.
        require
            non_negative: sum >= 0
        ensure
            one_more_deposit: deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
invariant
    consistent_balance: balance = all_deposits.total
end -- class interface ACCOUNT
```

(Eiffel) In the above code, if any contract is broken, the run time system traps to some handler for the error.

If there is no handler, the program exits, with the name of the broken part of the contract.

Wear a black hat...

... from time to time ...

Cautious, conservative programming is the main defense, but occasionally you need to put on your black hat, and find out what the attackers have discovered recently...

Testing and finding flaws

Still theoretical...

One aspect in engineering the security of software is that even one flaw in software may lead to a security issue. This leads to everything weighted on the side of the attacker:

- Lets say people find one flaw in 100 hours of testing/use.
- Let us also assume that there are $100,000$ errors in a large software product. A company employs testers for $100,000$ hours, and discover 1000 flaws.

Unfortunately:

An attacker discovers one flaw, in 100 hours, and the likelihood that the testers have found this specific error is only 1%.
The math favours the attacker.

S/W development for security

A sub species of S/W development...

Aspects of engineering the security of software may include extra work in areas such as design reviews and testing, along with administrative duties such as system and configuration management and post mortems.

Design and code review:

One study suggested that security flaws are found in a large software project at the following times:

- 17% found when doing system design inspection
- 19% found when doing component design inspection
- 15% found when looking at the code
- 30% found when doing integration testing
- 16% found afterwards during general testing.

Of course we have no idea how many errors are left in the code.