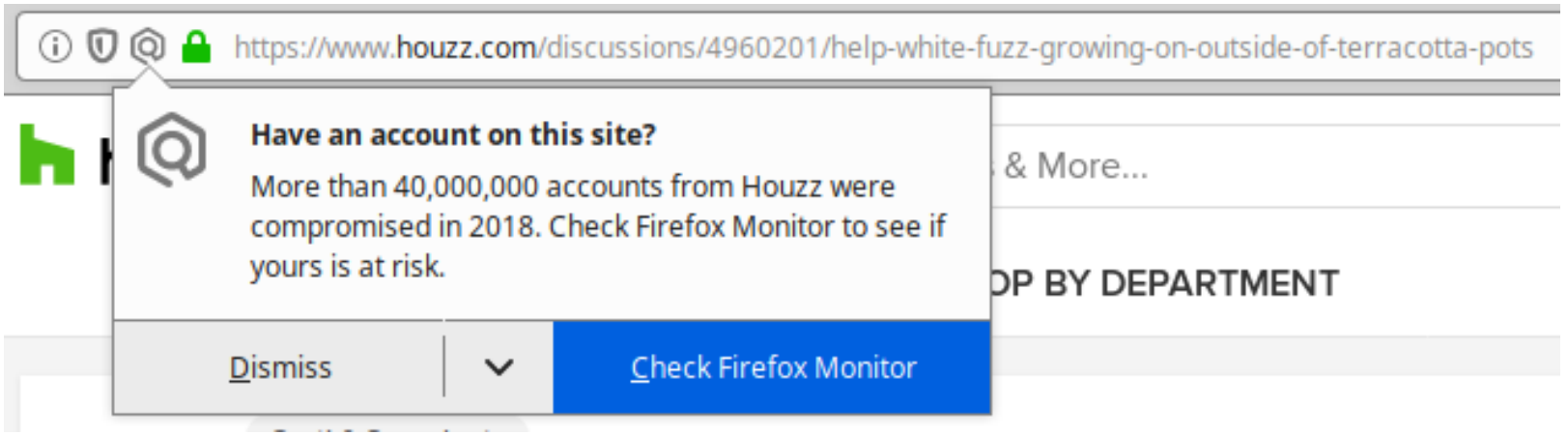
ECE 455: CYBERSECURITY

Lecture #3

Daniel Gitzel

In the news...

- **New browser feature warns about previous site breaches.**



Announcement

- **Reading will be posted for next week**
- **Quiz next week**
- **Lab 0.5 due next week**

REVIEW

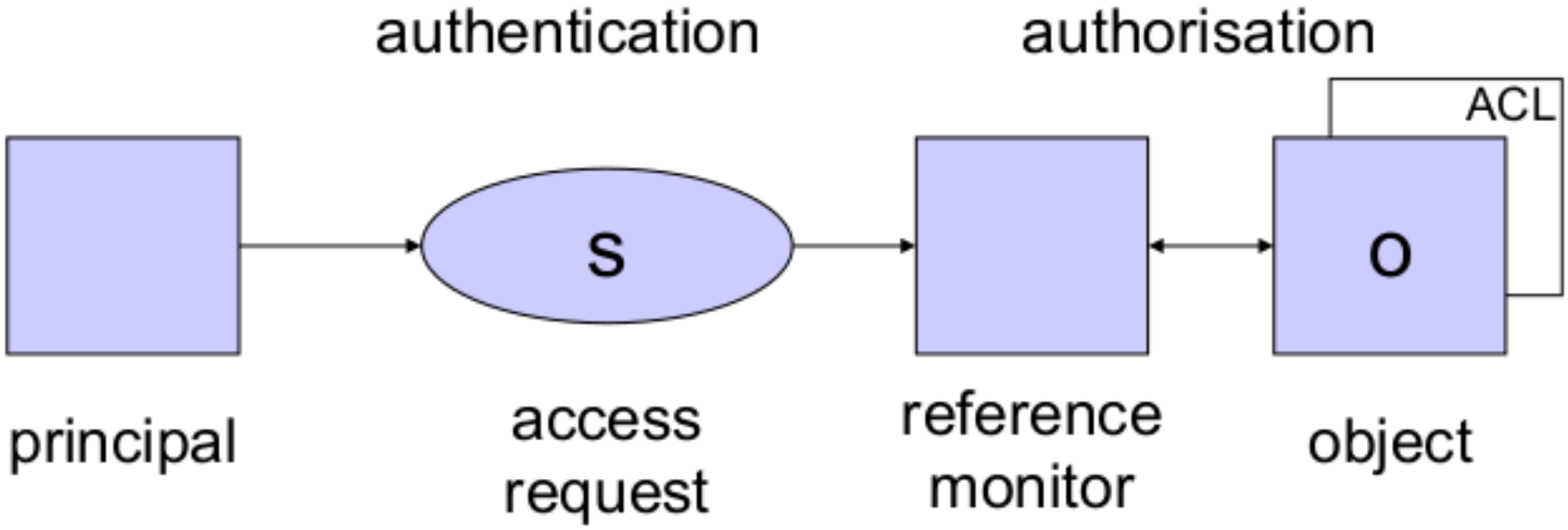
Introduction

- **User is authenticated**
  - I know who you are!
- **Who is allowed to do what?**
  - What privileges, permissions, power do you have?
- **Traditionally, consists of an operation performed on a resource**
  - read, write, execute on a file, directory, or port
- **Today, this can be more abstract**

Security Policies

- **Access control enforces operational security policies.**
- **A *policy* specifies who is allowed to do what.**
- **The active entity requesting access to a resource is called the *principal***
- **The resource access is requested for is called the *object*.**
- **Traditionally, policies refer to the requester's identity and decisions are *binary* (yes/no).**

Authentication vs. Authorization



B. Lampson, M. Abadi, M. Burrows, E. Wobber: Authentication in Distributed Systems: Theory and Practice, ACM Transactions on Computer Systems, 10(4), pages 265-310, 1992

Basic Terminology – Recap

***Subject/Principal*: Active entity – user or process.**

***Object*: Passive entity – file or resource.**

***Access operations*: basic memory access (read, write), method calls, push to network, etc.**

Comparable systems may use different access operations or attach different meanings to operations which appear to be the same.

Access Operations

- ***Access right***: right to perform an access or operation
- ***Permission***: typically a synonym for access right.
- ***Privilege***: typically a ***set of access rights*** given directly to roles like administrator, operator, ...
- These terms may have specific meanings in different systems.

Access Operations

- **On the most elementary level, a subject may**
  - observe an object, or
  - alter an object.
- **Some policies can be expressed with these *access modes*.**
- **A richer set of operations is more convenient.**

Access Control Structure

- **Policy is stored in an *access control structure*.**
  - Captures your desired access control policy.
  - You should be able to check that your policy has been captured correctly.
- **Access rights can be defined individually for each combination of subject and object.**
- **For large numbers of subjects and objects, such structures are cumbersome to manage; *intermediate levels of control* are preferable.**

Who Sets the Policy?

- **Security policies specify how principals are given access to objects.**
- **Responsibility for setting policy could be assigned to:**
  - the *owner* of a resource, who may decree who is allowed access; such policies are called *discretionary* as access control is at the owner's discretion.
  - a *system wide policy* decreeing who is allowed access; such policies are called *mandatory*.
- ***Warning*: other interpretations of discretionary and**
- **mandatory access control exist.**

REFERENCE MONITORS

Agenda

- **Reference monitor, security kernel, and TCB**
  - Placing the reference monitor
- **Status information & controlled invocation**
- **Security features in microprocessors**
  - Confused deputy problem
- **Memory management and access control**
- **Historic examples, to keep matters simple**

Security Mechanisms

- **How can computer systems enforce operational policies in practice?**
- **Questions that have to be answered:**
  - Where should access control be located? (Second Fundamental Design Decision)
  - Are there any additional security requirements your solution forces you to consider?
- **The following definitions are taken from the Orange Book.**

Reference Monitor (RM)

- **Reference monitor: an abstract machine that mediates all accesses to objects by subjects.**
- **Security Kernel: hardware, firmware, and software elements that implement the reference monitor.**
  - Must mediate all accesses, be protected from modification, and be verifiable as correct.
  - Part of the trusted computing base (TBC)

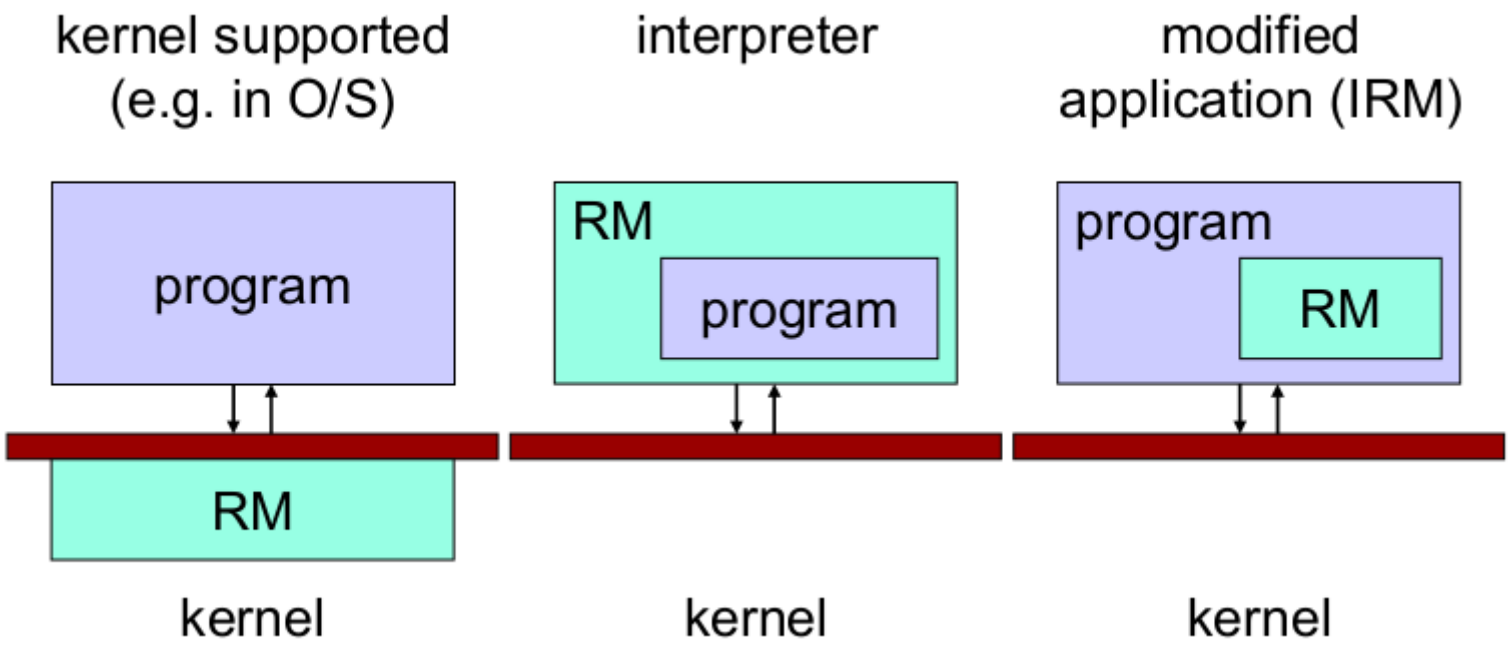
Trusted Computing Base (TCB)

- **The totality of protection mechanisms within a computer system.**
- **Consists of one or more components that together enforce a security policy over a product or system.**
  - Hardware, firmware, and software – the combination of which is responsible for enforcing a security policy.
- **TCB to correctness depends on the mechanisms within the TCB and on the correctness of security policy parameters.**

Placing the RM

- **Hardware**: access control mechanisms in microprocessors
- **OS kernel**: hypervisors, i.e. a virtual machine that emulates a host computer
- **OS**: access control in Unix and Windows
- **Services layer**: access control in database systems, Java Virtual Machine, .NET Common Language Runtime, or middleware architecture
- **Application**: security checks in the application code

RM – Design Choices



Operating System Integrity

- **Assume OS prevents unauthorized access to resources (as long as it works as intended).**
  - An attacker may try to disable the security controls by modifying the OS.
- **Now facing an integrity problem.**
  - The OS is not only the arbitrator of access requests
  - It is itself an object of access control.
- **New security policy: Users must not be able to modify the operating system.**
  - This generic security policy needs strong and efficient support.

Operating System Integrity

- **To make life more complicated, you have to address two competing requirements.**
  - Users should be able to use (invoke) the OS.
  - Users should not be able to misuse the OS.
- **Two important concepts commonly used to achieve these goals are:**
  - status information
  - controlled invocation, also called restricted privilege
- **These concepts can be used in any layer of an IT system, be it application software, OS, or hardware.**

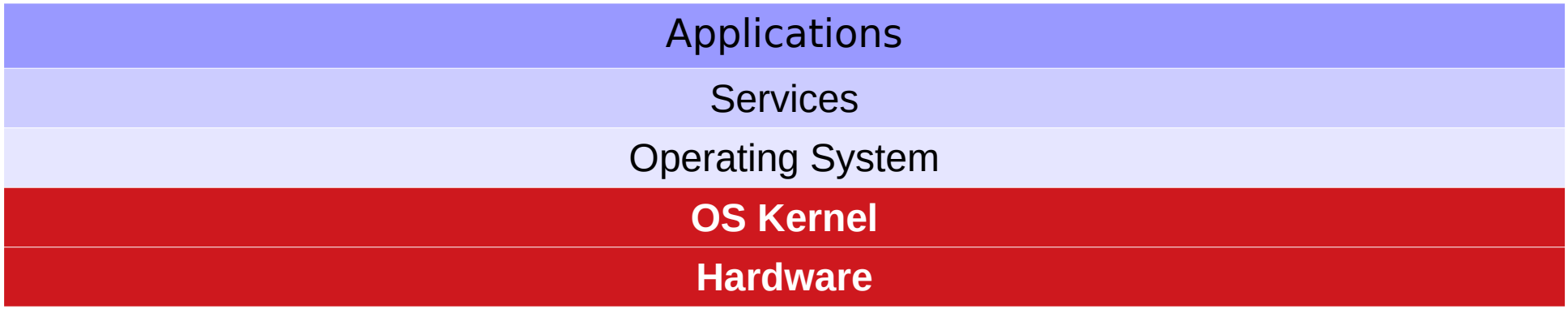
Modes of Operation

- **To protect itself, an OS must be able to distinguish**
  - computations ‘on behalf’ of the OS
  - computations ‘on behalf’ of a user.
- **Status flag allows system to work in different modes.**
  - Intel 80x86: two status bits and four modes
  - Unix distinguishes between user and superuser (root)
- **Example: stop users from writing directly to memory and corrupting the logical file structure**
  - OS grants write access to memory locations only if the processor is in supervisor mode.

Controlled Invocation

- **Example continued: A user wants to write to memory (requires supervisor mode)**
- **How should the system switch between modes?**
  - Changing the status bit to supervisor mode would give all supervisor privileges to the user
  - But doesn’t control on what the user actually does!
- **System should only perform a predefined set of operations in supervisor mode**
- **Return to user mode before handing control back to the user**
- **Let’s refer to this process as controlled invocation**

Core Security Mechanisms



Why Mechanisms at the Core?

- **For security evaluation at a higher level of assurance.**
  - To evaluate security, you must check that security mechanisms cannot be bypassed.
  - The more complex a system, the more difficult this check becomes.
  - At the core of a system you may find simple structures which are amenable to thorough analysis.
- **Security mechanisms in a given layer can be compromised from a layer below.**

Why Mechanisms at the Core?

- **Reduce performance overheads caused by security**
- **Processor performance depends on the right choice and efficient implementation of a generic set of operations that is most useful to the majority of users**
- **The same holds for security mechanisms**
  - Note: Some sources assume that TCBs and security kernels must enforce multi-level security policies

Computer Architecture

- **Simple computer architecture:**
  - central processing unit (CPU)
    - Instruction set
    - Logic design
    - Endianness
  - memory
    - Volatile and non-volatile
  - bus connecting CPU and memory and I/O
  - input/output devices

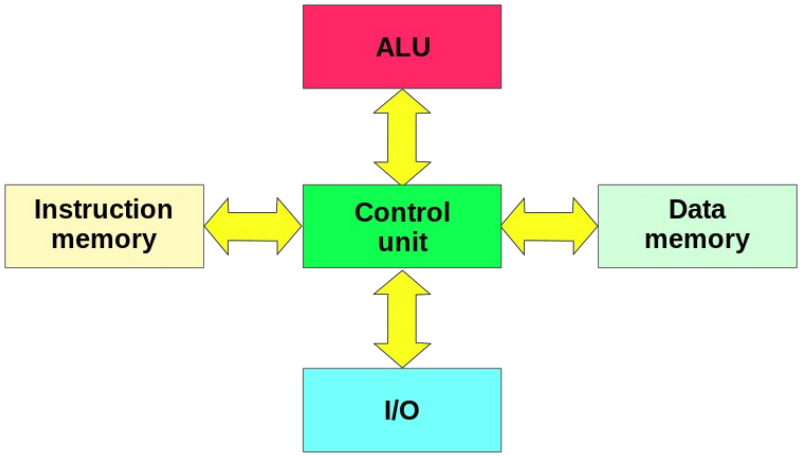
Computer Architecture

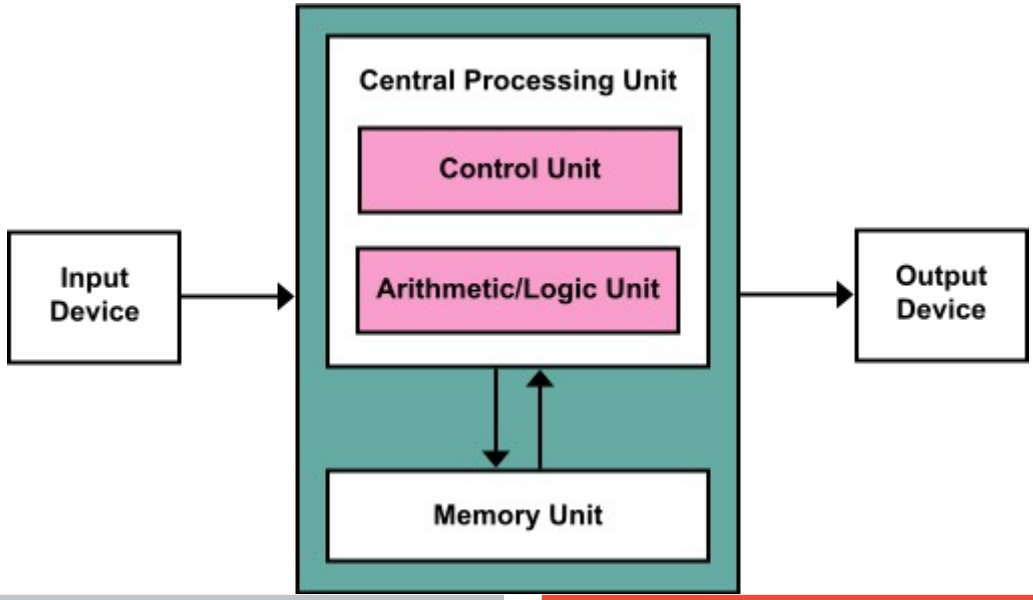
Harvard

- Separates instructions and data in memory

Von Neumann

- Combines instructions and data in memory





Core CPU Components

- **Registers: general purpose registers and dedicated registers like:**
  - program counter: points to the memory location containing the next instruction to be executed.
  - stack pointer: points to the top of the system stack.
  - status register: here CPU keeps essential state information.
- **Arithmetic Logic Unit (ALU):**
  - executes instructions given in a machine language
  - executing an instruction may also set bits in the status register.

Memory Structures

- **Security characteristics memory types**
- **RAM (random access memory)**
  - read/write memory; no guarantee of integrity or confidentiality.
- **ROM (read-only memory)**
  - provides integrity but not confidentiality; ROM may store (part of) the O/S.
- **EPROM (erasable & programmable read-only memory):**
  - could store OS or cryptographic keys; technologically more sophisticated attacks threaten security.
- **WROM (write-once memory):**
  - memory contents are frozen once and for all, e.g. by blowing a fuse placed on the write line; WROM could hold cryptographic keys or audit logs.

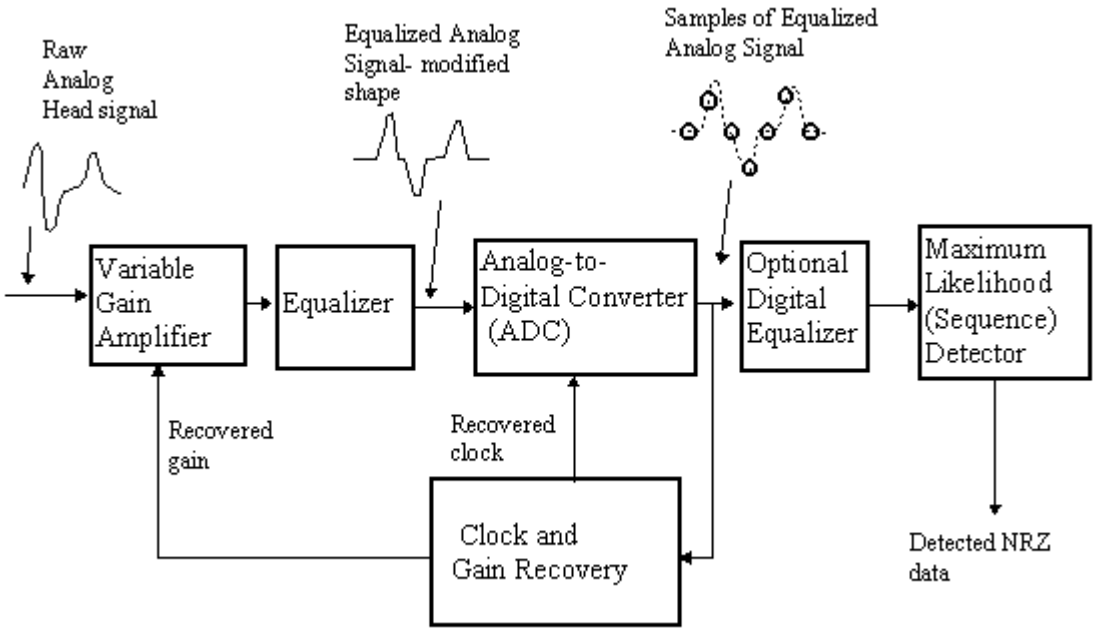
Memory Structures

- **Volatile memory loses its contents when power is switched off**
  - Memory contents still present after a short power loss.
  - Can be reconstructed by special electronic techniques if power has been switched off for some time.
  - To counter such attacks, memory has to be overwritten repeatedly with suitable bit patterns.
  - (see Guttmann and Noise-Predictive Maximum-Likelihood)
- **Non-volatile (permanent) memory keeps its content when power is switched off**
  - If attacker can directly access memory bypassing the CPU, cryptographic or physical measures are needed to protect sensitive data.
  - E.g., a light sensor in a tamper resistant module may detect an attempted manipulation and trigger the deletion of the data kept in the module.

Extraction Techniques

Cryogenically frozen RAM bypasses all disk encryption methods

Computer encryption technologies have all relied on one key assumption that RAM (Random Access Memory) is volatile and that all content is lost when power is lost. That key assumption is now being fundamentally challenged with a $7 can of compressed air and it's enough to give every security professional heart burn.



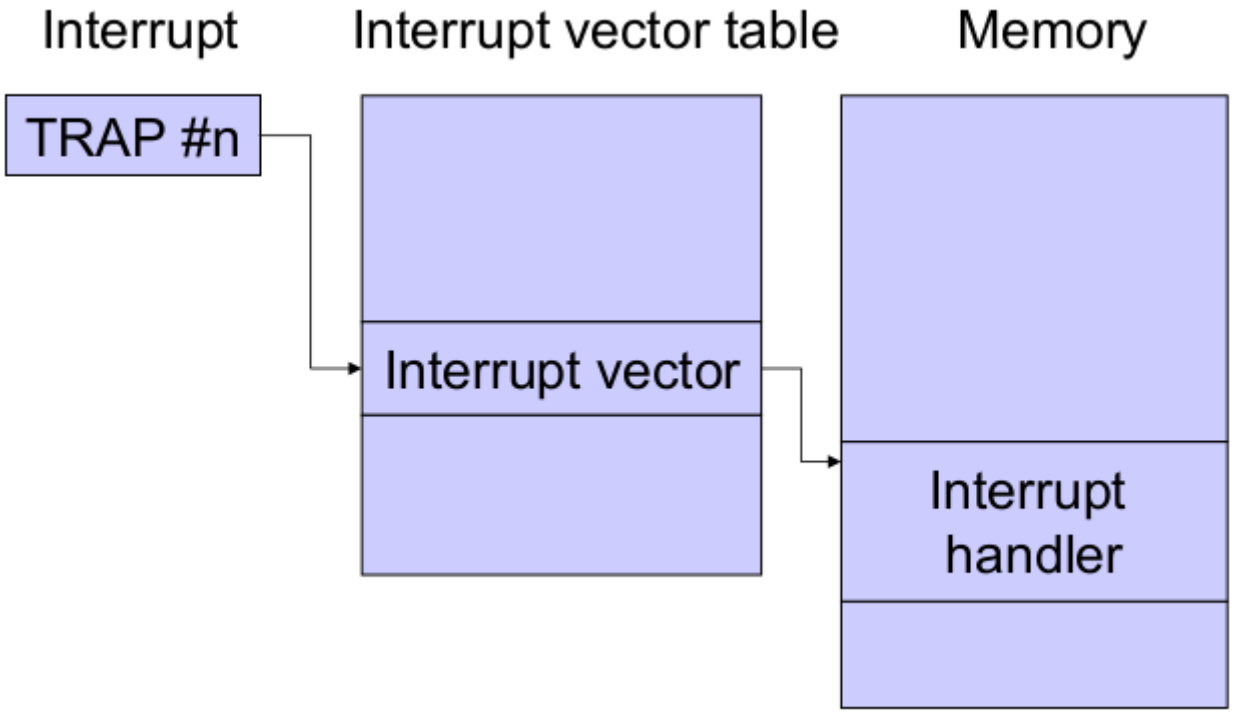
Processes and Threads

- **Process: a program in execution**
  - Consists of executable code, data, and the execution context, e.g. the contents of certain CPU registers
  - A process has its own address space and communicates with other processes only through OS primitives
  - Logical separation of processes as a basis for security
  - A context switch between processes can be an expensive operation
- **Threads: strands of execution within a process.**
  - Threads share an address space
  - Avoid the overheads of a full context switch
  - But they also avoid potential security controls
- **Processes and threads are important units of control for the OS, and for security. They are the 'subjects' of access control.**

Traps – Interrupts

- **CPU deals with interruptions of executions: exceptions, interrupts, and traps**
  - errors in the program, user requests, hardware failure, etc.
  - These terms refer to different types of events; we use trap as the generic term.
- **A trap is a special input to the CPU**
  - Includes an address (interrupt vector) in an interrupt vector table
  - Address is a program (interrupt handler) that deals with the condition specified in the trap.
- **When a trap occurs, the CPU saves its current state on the stack and then executes the interrupt handler.**
- **The interrupt handler has to restore the CPU to a proper state, e.g. by clearing the supervisor status bit, before returning control to the user.**

Interrupt Vectors



Interrupting Interrupts

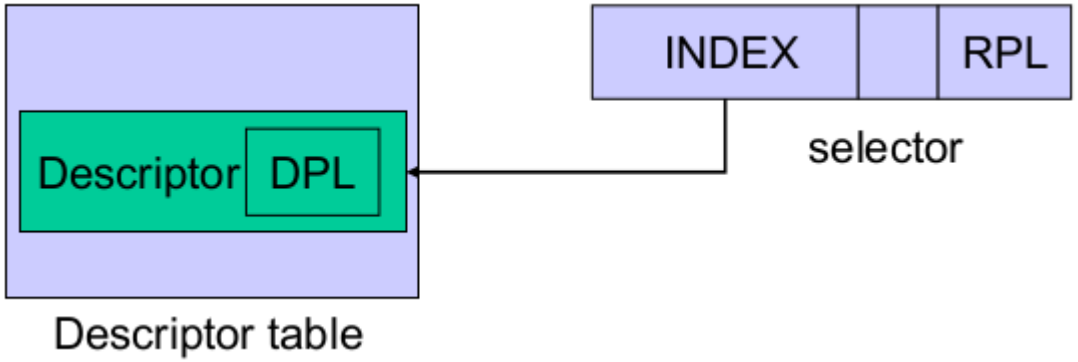
- **A new interrupt may arrive while the CPU deals with a current interrupt, so the CPU may have to interrupt the current interrupt handler.**
- **Improper handling of such a situation can cause security failures:**
  - Typing CTRL-C (user interrupts) so the CPU returns to the OS prompt with the status bit of the current process.
  - A user could then enter supervisor mode by interrupting the execution of an O/S call.
- **The interrupt table is a particularly interesting point of attack and has to be protected adequately.**
- **Redirecting pointers is an efficient way of compromising the integrity of the OS.**

Example: Intel 80x86

- **Support for access control at machine language level based on protection rings.**
- **Two-bit field in the status register: four privilege levels**
  - Unix, Windows use levels 0 (O/S) and 3 (user).
- **Privilege levels can only be changed through POPF.**
  - Processes can only access objects in their ring or in outer rings
  - Processes can invoke subroutines only within their ring
  - Processes need gates to execute procedures in an inner ring.
- **Information about system objects like memory segments, access control tables, or gates is stored in descriptors.**
- **The privilege level of an object is stored in the descriptor privilege level (DPL) field of its descriptor.**

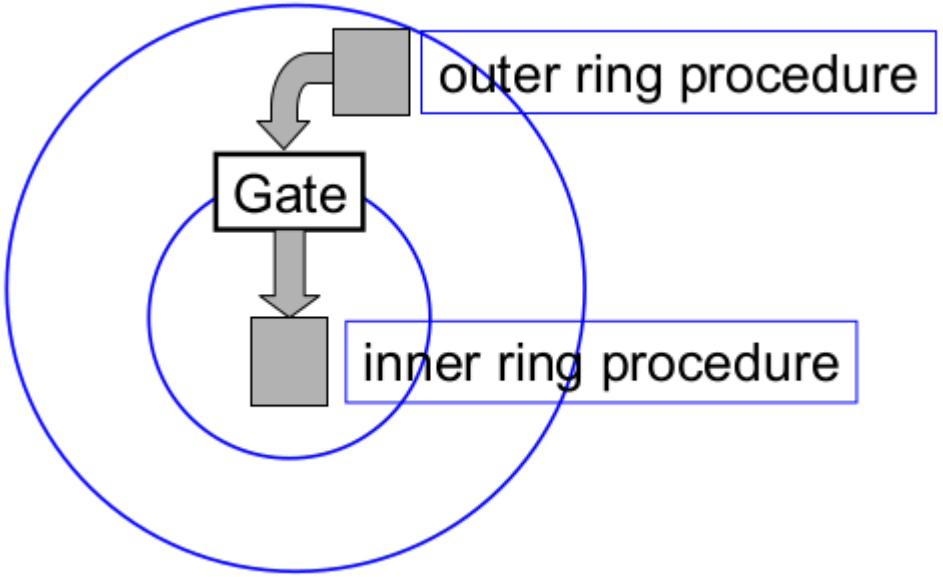
Intel 80x86 - Access Control

- **Descriptors held in descriptor table; accessed via selectors.**
- **Selector: 16-bit field, contains index for the object's entry in the descriptor table and a requested privilege level (RPL) field; only OS has access to selectors.**
- **Current privilege level (CPL): code segment register stores selector of current process; access control decisions can be made by comparing CPL (subject) and DPL (object).**


INDEX
RPL
selector
Descriptor
DPL
Descriptor table

Intel 80x86: Controlled Invocation

- **Gate: system object pointing to a procedure**
  - The gate has a privilege level different from that of the procedure it points to.
- **Allow execute-only access to procedures in an inner ring.**



Intel 80x86: Controlled Invocation

- **A subroutine call saves state information about the calling process and the return address on the stack.**
  - Should this stack be in the inner ring? Violates the security policy forbidding write to an inner ring.
  - Should this stack be in the outer ring? The return address could be manipulated from the outer ring.
- **Therefore, part of the stack (how much is described in the gate's descriptor) is copied to a more privileged stack segment.**

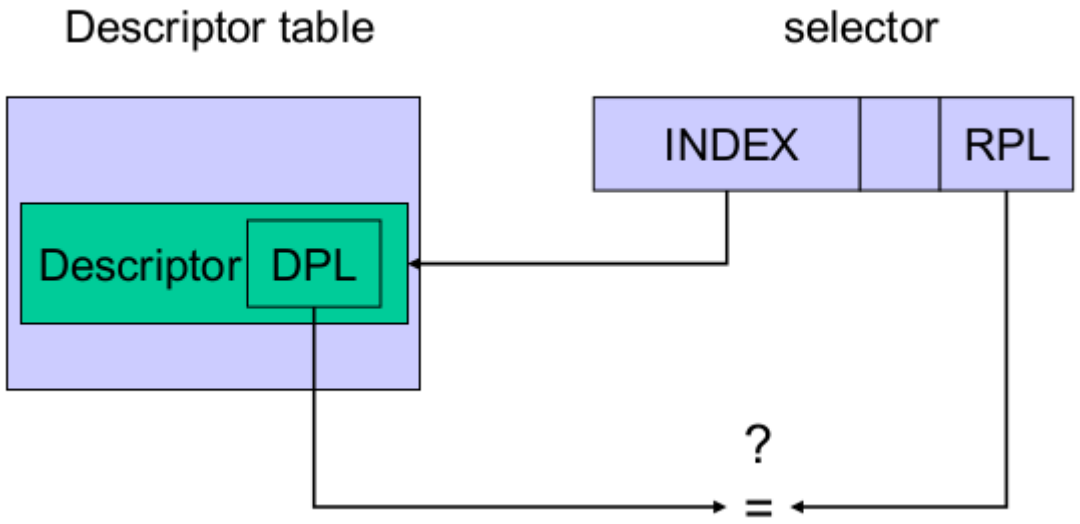
A Loophole?

- **Invoking a subroutine through a gate**
  - CPL changes to the level of the code the gate is pointing to
  - On returning from the subroutine, the CPL is restored to that of the calling process.
- **The outer-ring process may ask the inner-ring procedure to copy an inner ring object to the outer ring**
- **This will not be prevented by any of the mechanisms presented so far**
- **Known as luring attack, or as confused deputy**
- **problem.**

Remedy

- To take into account the level of the calling process, use the adjust privilege level (ARPL) instruction.
- This instruction changes the RPL fields of all selectors to the CPL of the calling process.
- The system then compares the RPL (in the selector) and the DPL (in the descriptor) of an object when making access control decisions.

Comparing RPL and DPL



Security Mechanisms in OS

- **OS manages access to data and resources; multitasking OS interleaves execution of processes belonging to different users. It has to**
  - separate user space from OS space,
  - logically separate users,
  - restrict the memory objects a process can access.
- **Logical separation of users at two levels:**
  - file management, deals with logical memory objects
  - memory management, deals with physical memory objects
- **For security, this distinction is important.**

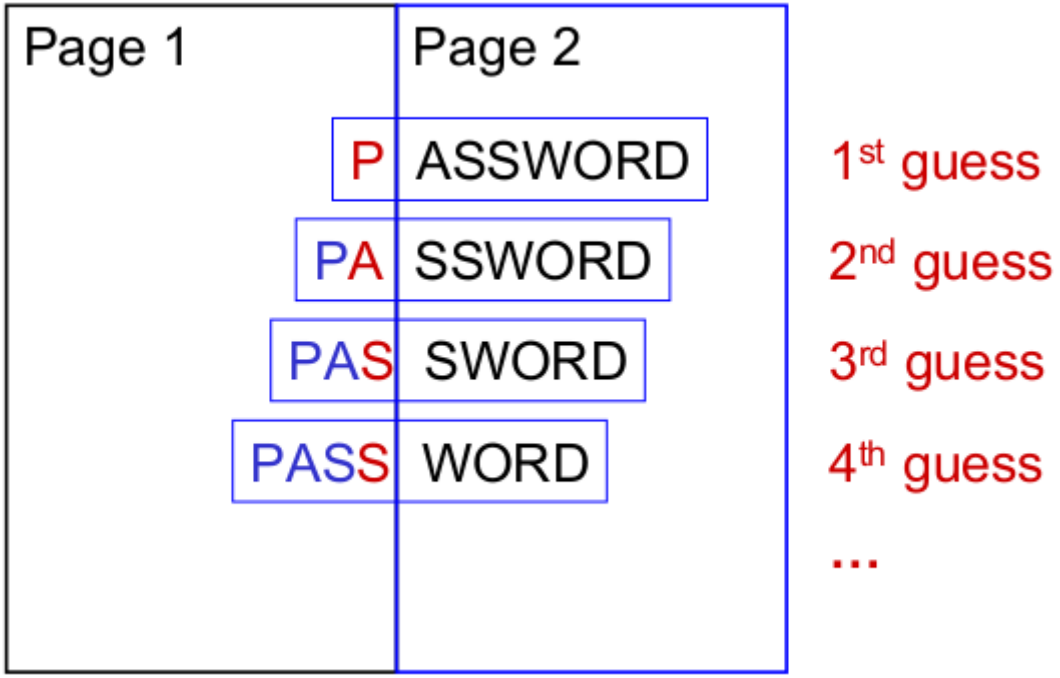
Segments and Pages

- **Segmentation divides memory into logical units of variable lengths.**
  - A division into logical units is a good basis for enforcing a security policy.
  - Units of variable length make memory management more difficult.
- **Paging divides memory into pages of equal length.**
  - Fixed length units allow efficient memory management.
  - Paging is not a good basis for access control as pages are not logical units.
  - One page may contain objects requiring different protection. Page faults can create a covert channel

A Covert Channel

- **Process accesses a logical object stored on more than one page, a page fault occurs whenever a new page is requested**
- **A covert channel exists if page faults are observable.**
- **Password entered is compared character by character with the reference password stored in memory**
  - Access is denied the moment an incorrect match is found.
- **If a password is stored across a page boundary, then observing a page fault indicates that the piece of the password on the first page has been guessed correctly**
  - If the attacker can control where the password is stored on the page, password guessing becomes easy

Exploiting the Covert Channel



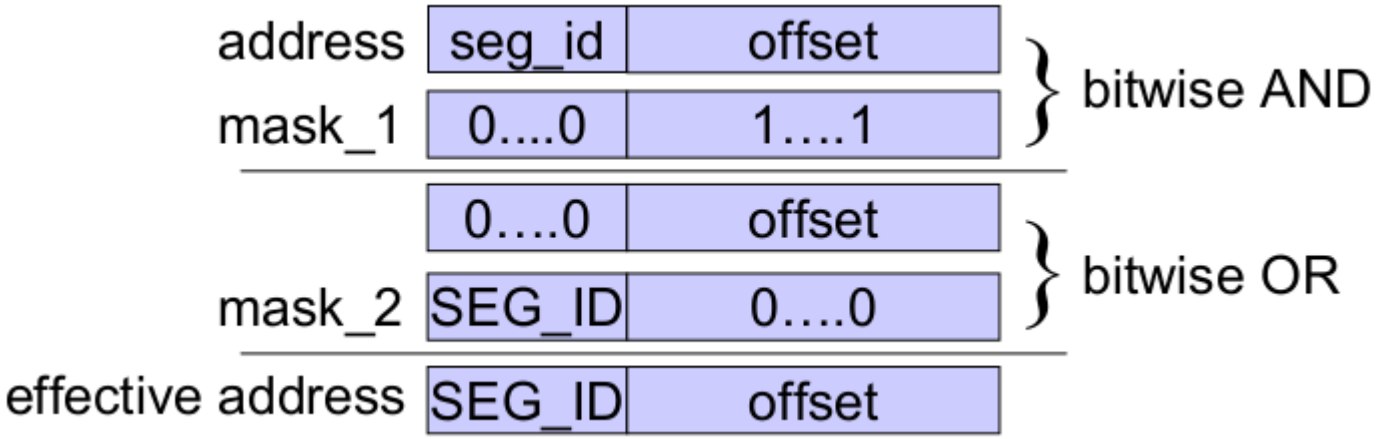
Memory Protection

- **OS controls access to data objects in memory.**
- **A data object is represented by a collection of bits stored in certain memory locations.**
- **Access to a logical object is ultimately translated into access operations at machine language level.**
- **Three options for controlling access to memory:**
  - operating system modifies the addresses it receives from user processes;
  - operating system constructs the effective addresses from relative addresses it receives from user processes;
  - operating system checks whether the addresses it receives from user processes are within given bounds.

Address Sandboxing (Modification)

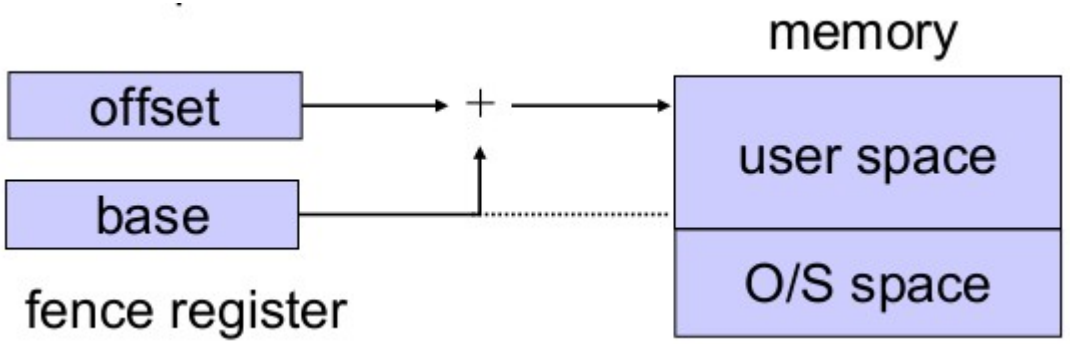
- **Address consists of segment identifier and offset.**
- **When the operating system receives an address, it sets the correct segment identifier as follows:**
  - Bitwise AND of the address with mask_1 clears the segment identifier

    Bitwise OR with mask_2 sets the segment identifier to the intended value SEG_ID.



Relative Addressing

- Clever use of addressing modes can keep processes out of forbidden memory areas.
- **Fence registers**: base register addressing keeps users out of OS space; fence register points to top of user space.
- **Bounds register** define the bottom of the user space.
- Base and bounds registers allow to **separate program from data space**.



Function codes

- **Motorola 68000 function codes**
  - Indicate processor status
  - Address decoder may select between
    - user and supervisor memory
    - data and programs

FC2	FC1	FC0	
0	0	0	(undefined,reserved)
0	0	1	user data
0	1	0	user program
0	1	1	(undefined,reserved)
1	0	0	(undefined,reserved)
1	0	1	supervisor data
1	1	0	supervisor program
1	1	1	interrupt acknowledge

General Lessons

- **Distinguishing between data and programs is a useful security feature**
  - providing a basis for protecting programs from modification
  - memory has been divided into different regions
- **Access control can then refer to the location a data object or program comes from**
- **This can serve as a first example for location based access control**
- **Distributed systems or computer networks may use location based access control at the level of network nodes**

Tagged architectures

- **Tagged architectures indicate type of each memory object**
  - Each word of memory is a tagged union
  - Pioneered in Soviet Elbrus super computers ~1973

Executable space protection

- **Hardware based execution prevention**
- **No-execute bit (NX bit) AMD64 ISA**
  - Intel’s XD bit or AMD’s EVP or ARM’s XN bit
  - NX bit refers to bit number 63 (the most significant bit) of a 64-bit entry in the page table.
  - If set to 1, code cannot be executed from that page
  - It is only available with the long mode (64-bit mode), but not x86's original 32-bit page table format
- **Processor will refuse to execute code in marked memory regions**

Summary

- **Security policies can be enforced in any layer of a computer system.**
- **Mechanisms at lower layers are more generic and are universally applied to all “applications” above, but might not quite match the requirements of the application.**
- **Mechanisms at upper layers are more application specific, but applications have to be secured individually.**
- **This fundamental dilemma is a recurring theme in information security.**

BUFFER OVERFLOWSSSSSSSSSSSSSSS

SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS

Segmentation fault

Attacks on Memory Buffers

- **Buffer is a predefined data storage area inside computer memory (stack or heap)**
- **Typical situation:**
  - A function takes some input that it writes into a pre-allocated buffer.
  - The developer forgets to check that the size of the input isn’t larger than the size of the buffer.
  - Uh oh.
    - “Normal” bad input: crash
    - “Adversarial” bad input : take control of execution

Stack Buffers

- **Suppose Web server contains this function**

```
void func(char *str) {
        char buf[126];
        ...
        strcpy(buf,str);
        ...
}
```

- **No bounds checking on strcpy()**
- **If str is longer than 126 bytes**
  - Program may crash
  - Attacker may change program behavior

Example: Changing Flags

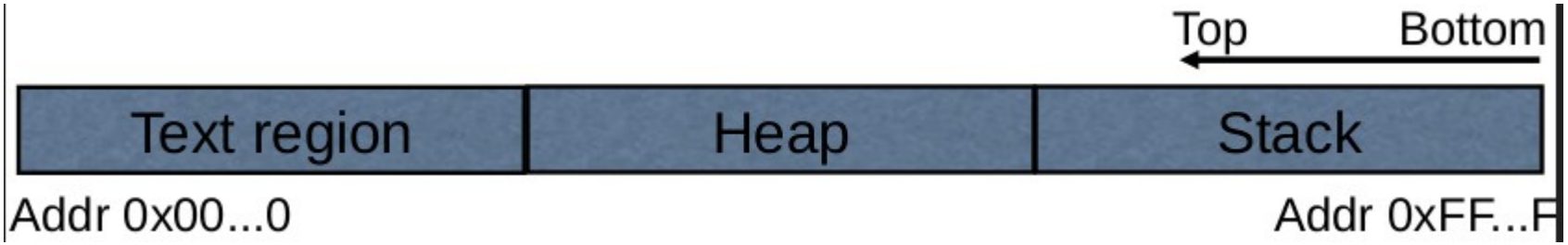
- **Suppose Web server contains this function**

```
void func(char *str) {
        char buf[126];
        ...
        strcpy(buf,str);
        ...
}
```

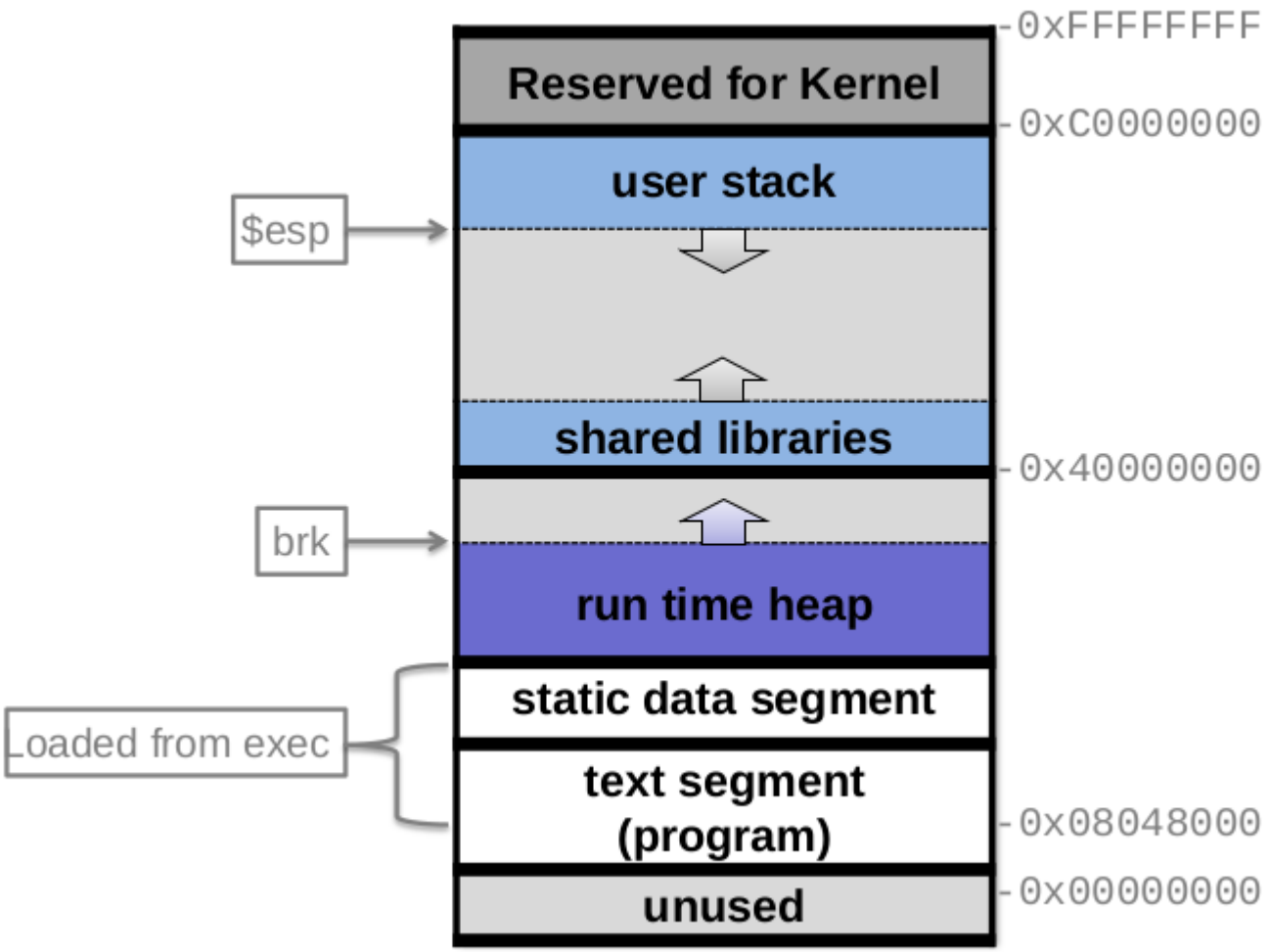
- **Authenticated variable non-zero when user has extra privileges**
- **Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd**

Memory Layout

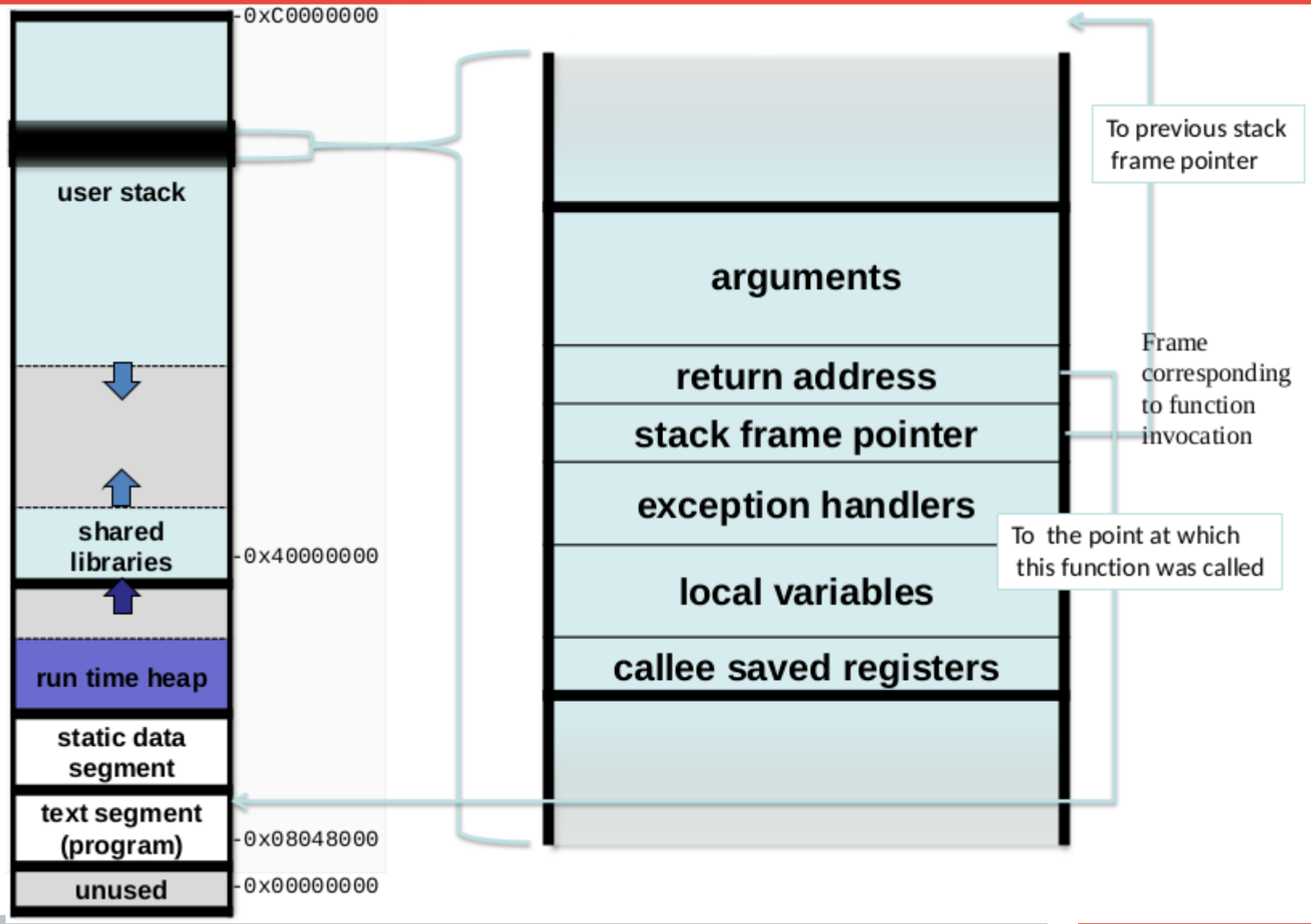
- **Text region: Executable code of the program**
- **Heap: Dynamically allocated data**
- **Stack: Local variables, function return addresses; grows and shrinks as functions are called and return**



Linux (32-bit) process memory layout



Stack Frame


0xC0000000
user stack
shared libraries
0x40000000
run time heap
static data segment
text segment (program)
0x08048000
unused
0x00000000
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
To previous stack frame pointer
Frame corresponding to function invocation
To the point at which this function was called

Stack Buffers

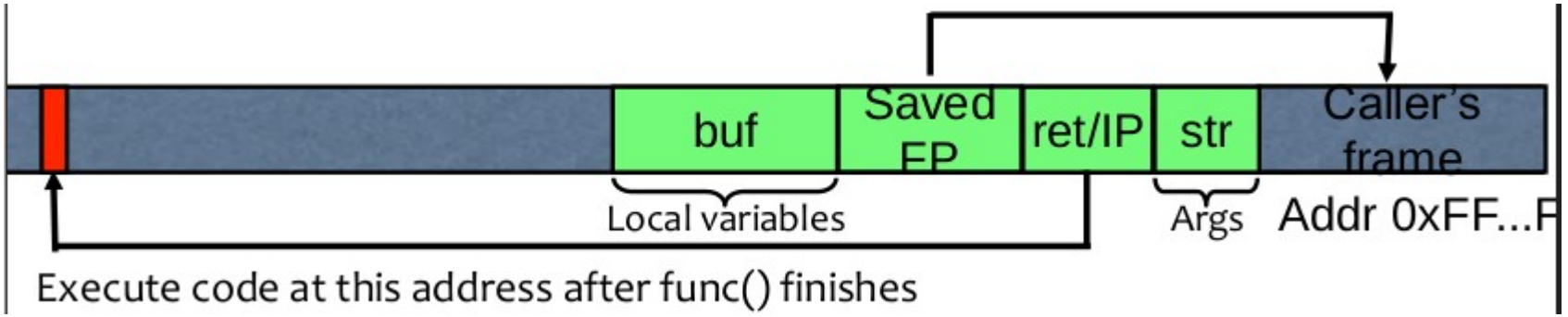
- **Suppose Web server contains this function**

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
}
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- **When this function is invoked, a new frame (activation record) is pushed onto the stack.**



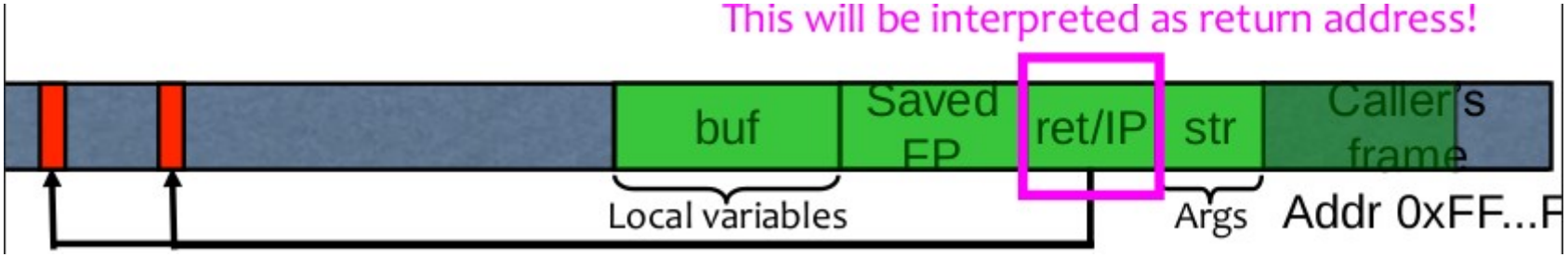
What if Buffer is Overstuffed?

- **Memory pointed to by str is copied onto stack...**

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
    }
```

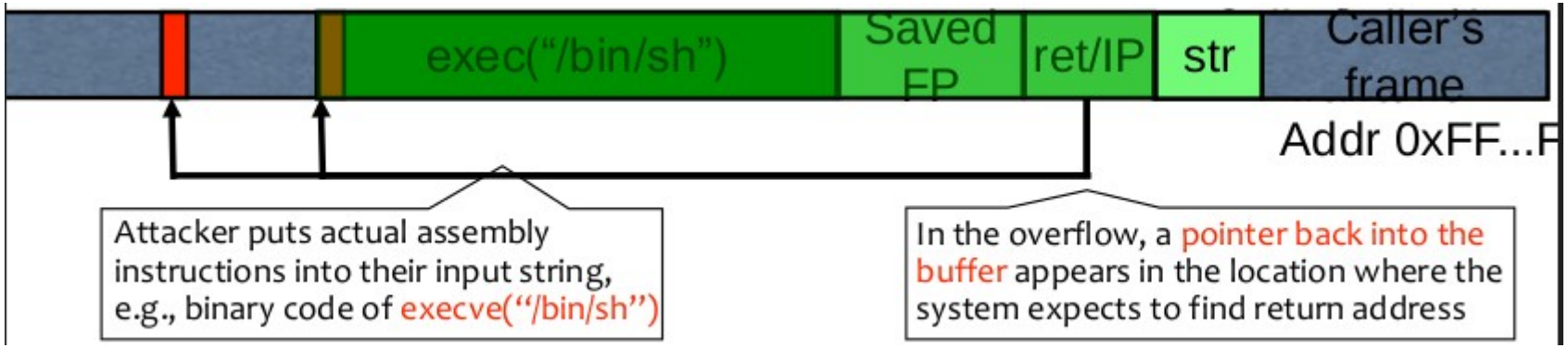
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- **If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.**



Executing Attack Code

- **Suppose buffer contains attacker-created string**
  - For example, str points to a string received from the network as the URL



- **When function exits, code in the buffer will be executed, giving attacker a shell ("shellcode")**
  - Root shell if the victim program is setuid root

Buffer Overflows Can Be Tricky…

- **Overflow portion of the buffer must contain correct address of attack code in the RET position**
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
  - Otherwise application will (probably) crash with segfault
  - Attacker must correctly guess in which stack position their buffer will be when the function is called

Problem: No Bounds Checking

- **strcpy does not check input size**
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of area allocated to buf
- **Many C library functions are unsafe**
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, ...)
  - printf(const char *format, ...)

Does Bounds Checking Help?

- **strncpy(char *dest, const char *src, size_t n)**
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
  - Programmer has to supply the right value of n
- **Potential overflow in htpasswd.c (Apache 1.3):**

```
strcpy(record,user);
strcat(record,":");
strcat(record,cpw);
```

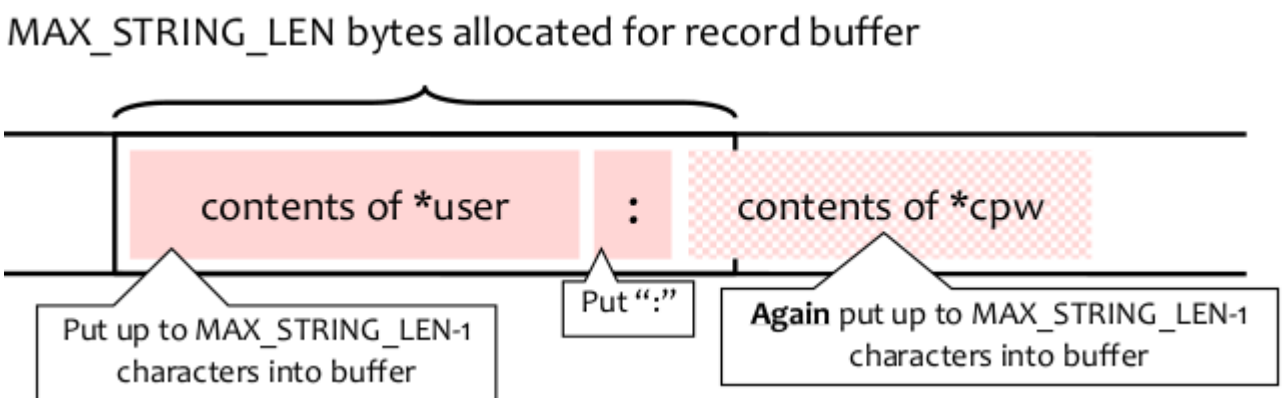
Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

```
strncpy(record,user,MAX_STRING_LEN-1);
strcat(record,":")
strncat(record,cpw,MAX_STRING_LEN-1);
```

Misuse of strncpy in htpasswd “Fix”

- **Published “fix” for Apache htpasswd overflow:**
  - MAX_STRING_LEN bytes allocated for record buffer

```
strncpy(record,user,MAX_STRING_LEN-1);
strcat(record,":")
strncat(record,cpw,MAX_STRING_LEN-1);
```



What About This?

- **Home-brewed range-checking string copy**

```
void mycopy(char *input) {
     char buffer[512]; int i;

     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        mycopy(argv[1]);
}
```

What About This?

- **Home-brewed range-checking string copy**

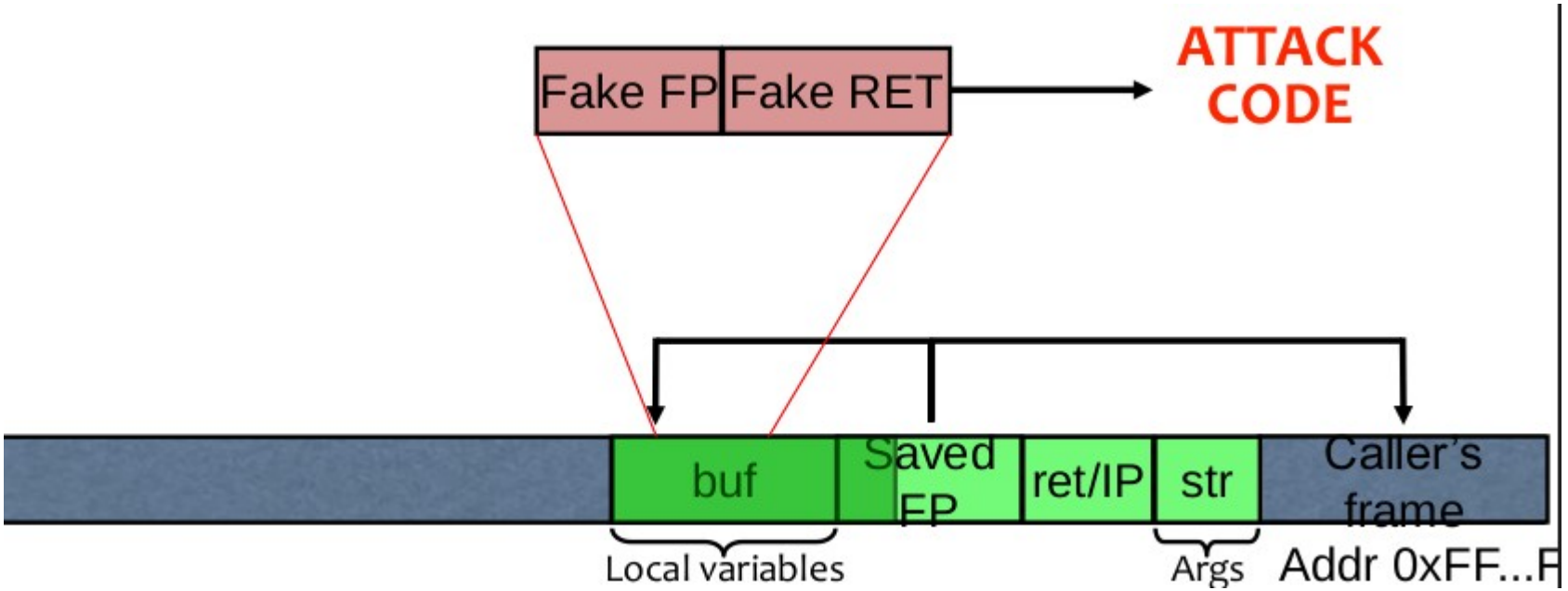
```
void mycopy(char *input) {
     char buffer[512]; int i;

     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        mycopy(argv[1]);
}
```

This will copy 513 characters into buffer. Oops!

- **1-byte overflow: can't change RET, but can change pointer to previous stack frame…**

Frame Pointer Overflow



Other Overflow Targets

- **C uses function pointers for callbacks: if pointer to F is stored in memory location P, then one can call F as (*P)(...)**

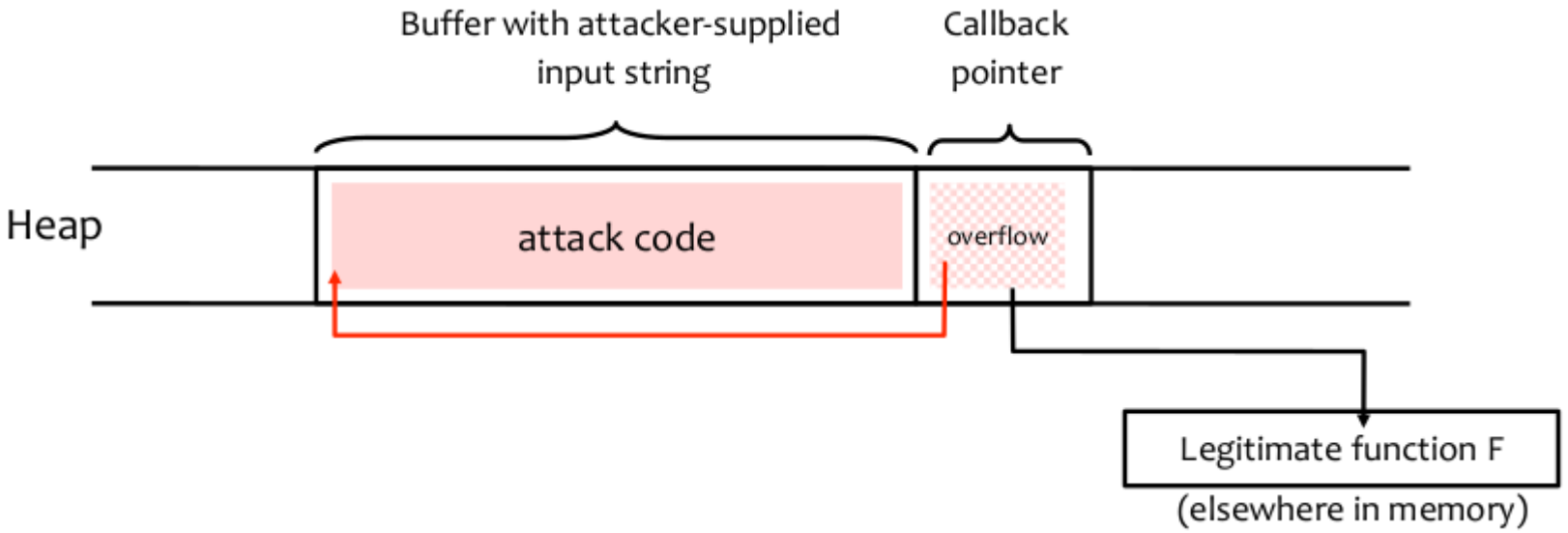
```c
#include <stdio.h>  /* for printf */
#include <string.h> /* for strchr */

double cm_to_inches(double cm) {
    return cm / 2.54;
}

// "strchr" is part of the C string handling (i.e., no need for declaration)
// See https://en.wikipedia.org/wiki/C_string_handling#Functions

int main(void) {
    double (*func1)(double) = cm_to_inches;
    char * (*func2)(const char *, int) = strchr;
    printf("%f %s", func1(15.0), func2("Wikipedia", 'p'));
    /* prints "5.905512 pedia" */
    return 0;
}
```

Other Overflow Targets



Variadic Functions in C

- **A function variable number of arguments**

```c
double average(int count, ...)
{
    va_list ap;
    int j;
    double sum = 0;

    va_start(ap, count); /* Requires the last fixed parameter (to get the address) */
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, int); /* Increments ap to the next argument. */
    }
    va_end(ap);

    return sum / count;
}

int main(int argc, char const *argv[])
{
    printf("%f\n", average(3, 1, 2, 3) );
    return 0;
}
```

```
printf("hello, world");
printf("length of '%s' = %d\n", str, str.length());
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by special % characters

- **Special functions va_start, va_arg, va_end compute arguments at run-time**
  - va_list is an argument pointer (stack pointer)

```c
double average(int count, ...)
{
    va_list ap;
    int j;
    double sum = 0;

    va_start(ap, count); /* Requires the last fixed parameter (to get the address) */
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, int); /* Increments ap to the next argument. */
    }
    va_end(ap);

    return sum / count;
}

int main(int argc, char const *argv[])
{
    printf("%f\n", average(3, 1, 2, 3) );
    return 0;
}
```

Format Strings in C

- **Proper use of printf format string:**

```
int foo = 1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

- **Sloppy use of printf format string:**

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buf contains "%d"?

Viewing Memory

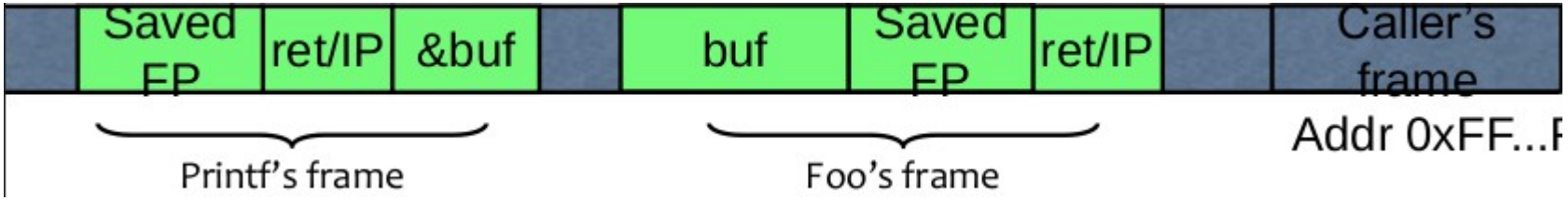
- **Tell printf to output data on stack (in hex format)**
  - printf(“Here is an int: %x”,i);
- **What if printf does not have an argument?**
  - char buf[16]=“Here is an int: %x”; printf(buf);
  - argument pointer is interpreted and printed as int
- **Or?**
  - char buf[16]=“Here is a string: %s”; printf(buf);
  - argument pointer is interpreted and printed as string

%n format specifier

- **%n is a special format specifier**
- **Does not print anything**
- **Causes printf() to write to the variable pointed by the corresponding argument**
- **A value equal to the number of characters that have been printed by printf() before the occurrence of %n**
  - printf("blah %n blah\n", &val); printf(“val=%d”, val);
  - Prints “val=5” (5 chars blah + “ “ before %n)

Writing Stack with Format Strings

- **What if printf does not have an argument?**
  - char buf[16]="blah %n blah"; printf(buf);
  - printf's internal stack pointer will be interpreted as address into which the number of characters will be written
- **printf(buf) is vulnerable!**
- **Foo() {char buf[128] = ...; printf(buf);}**



Using %n to Overwrite Return Address

- When %n “prints”, make sure the location under printf’s stack pointer contains address of RET; %n will write the number of characters in attackString into RET
- Do this with the right format string to overwrite RET byte-by-byte
- RET to your attack code, possibly in the crafted “buf”