

Security Architectures Inside The Programming Language

Frank Piessens
(Frank.Piessens@cs.kuleuven.be)

Overview

- Introduction
- Illustrating the risks of unsafe languages
- Safety and type soundness
- Sandboxing
- Conclusion

Introduction

- CERT Advisory Feb 2006:
“The Microsoft Windows Media Player plug-in for browsers other than Internet Explorer contains a buffer overflow, **which may allow a remote attacker to execute arbitrary code.**” (CVE-2006-005)
- This is one (of the many) examples of a vulnerability that is exploitable by a *code injection attack*
- Code injection attacks are mainly a risk for code written in **unsafe** languages

Overview

- Introduction
- Illustrating the risks of unsafe languages
- Safety and type soundness
- Sandboxing
- Conclusion

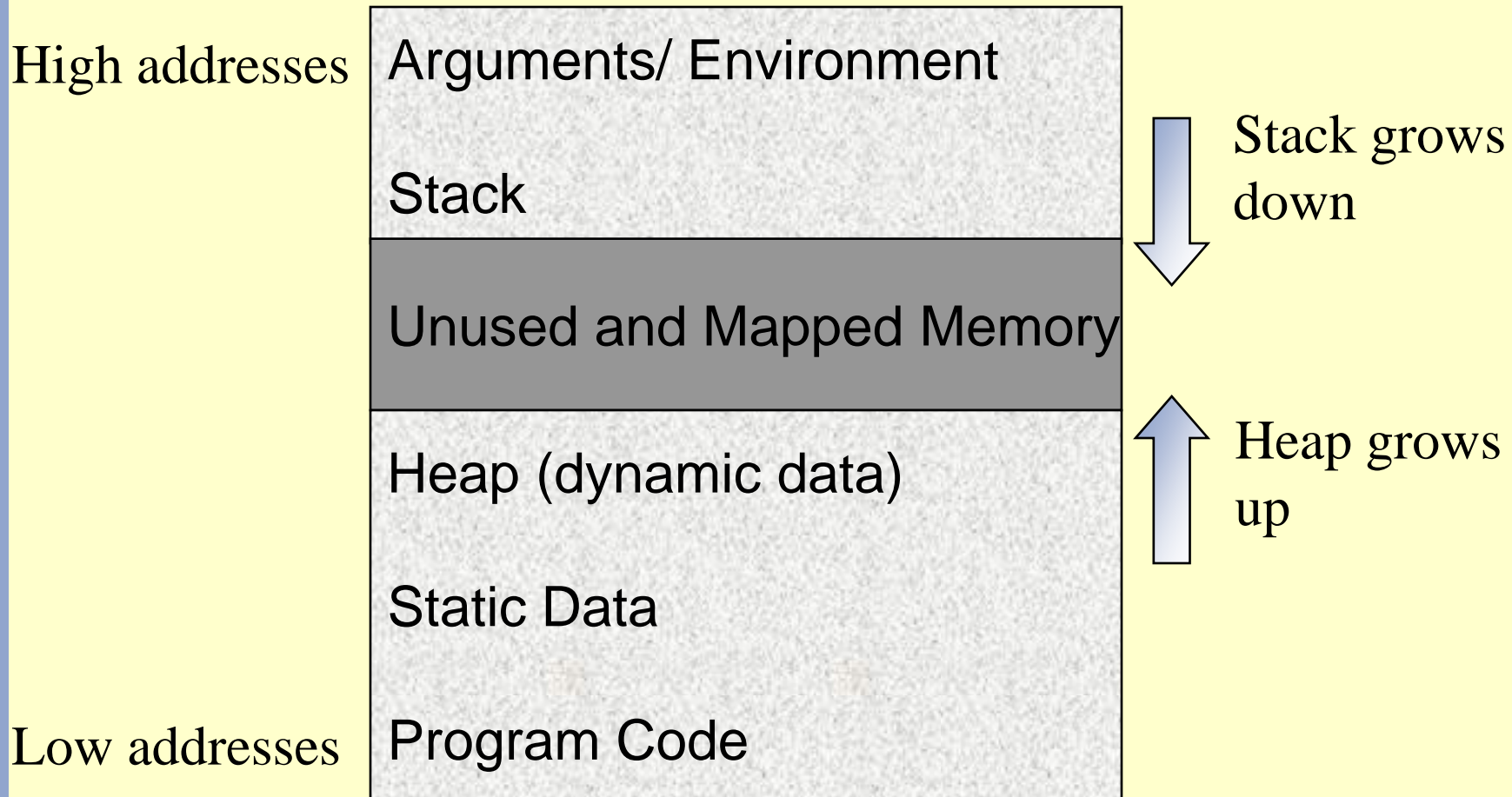
Memory management in C/C++

- Memory can be allocated in many ways in C/C++
 - Automatic (local variables in functions)
 - Static (global variables)
 - Dynamic (malloc and new)
- Programmer is responsible for:
 - Appropriate use of allocated memory
 - E.g. bounds checks, type checks, ...
 - Correct de-allocation of memory

Memory management in C/C++

- Memory management is very error-prone
- Some typical bugs:
 - Writing past the bound of an array
 - Dangling pointers
 - Double freeing
 - Memory leaks
- For efficiency, practical C/C++ implementations don't detect such bugs at run time
 - The language definition states that behavior of a buggy program is *undefined*

Process memory layout



Attacking unsafe code

- To do a code injection attack, an attacker must:
 - Find a bug in the program that can break memory safety
 - Find an interesting memory location to overwrite
 - Get attack code in the process memory space

Bugs that can break memory safety

- Writing past the end of an array (*buffer overrun or overflow*)
- Dereference a dangling pointer
- Use of a dangerous API function
 - That internally overflows a buffer
 - E.g. strcpy(), gets()
 - That is implemented in assembly in an intrinsically unsafe way
 - E.g. printf()

Interesting memory locations

- Code addresses or function pointers
 - Return address of a function invocation
 - Function pointers in the virtual function table
 - Program specific function pointers
- Pointers where the attacker can control what is written when the program dereferences the pointer
 - Indirect pointer overwrite: first redirect the pointer to another interesting location, then write the appropriate value

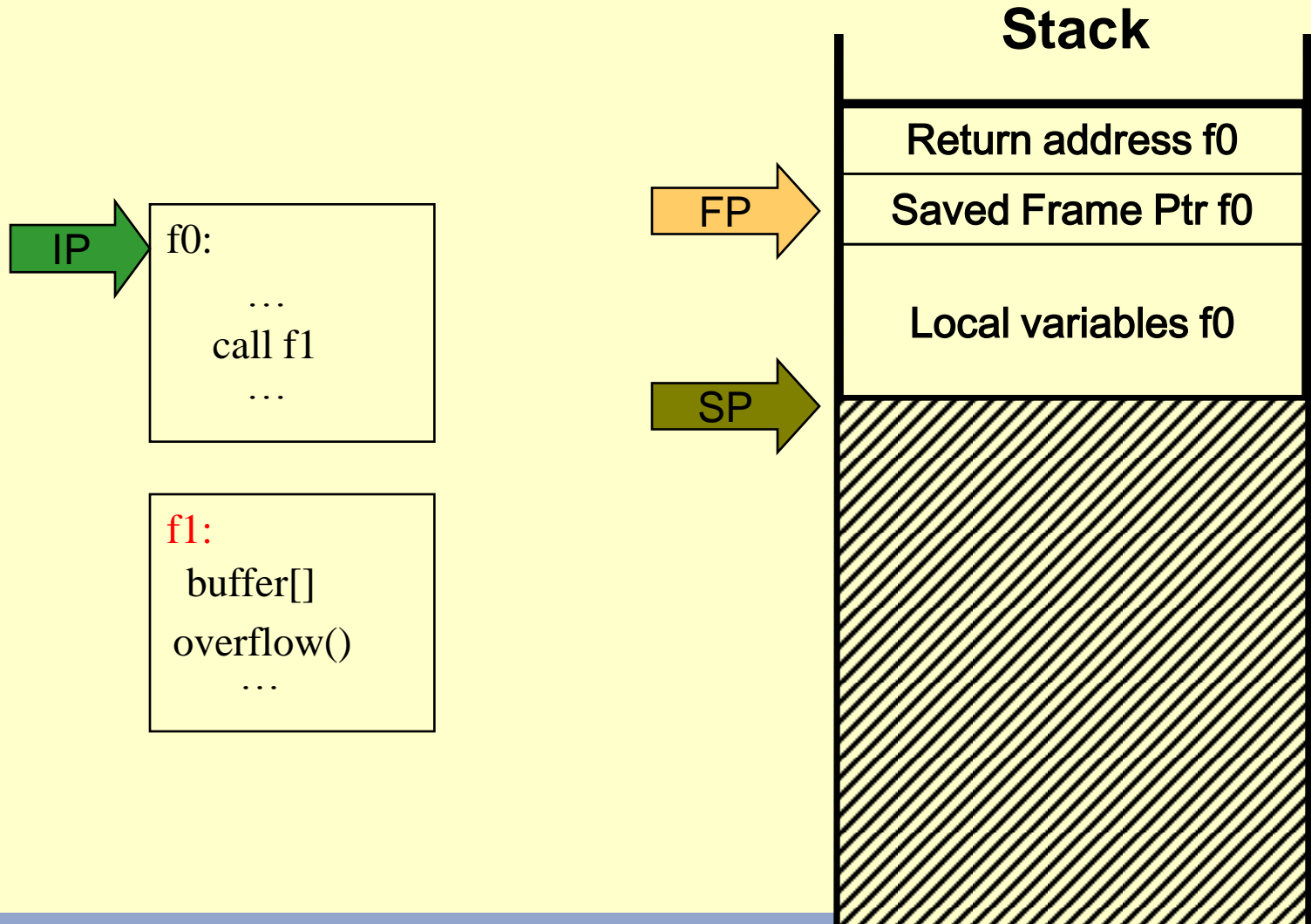
Some example attacks

- Stack-based buffer overrun
- Heap-based buffer overrun
- Exploiting a format string vulnerability

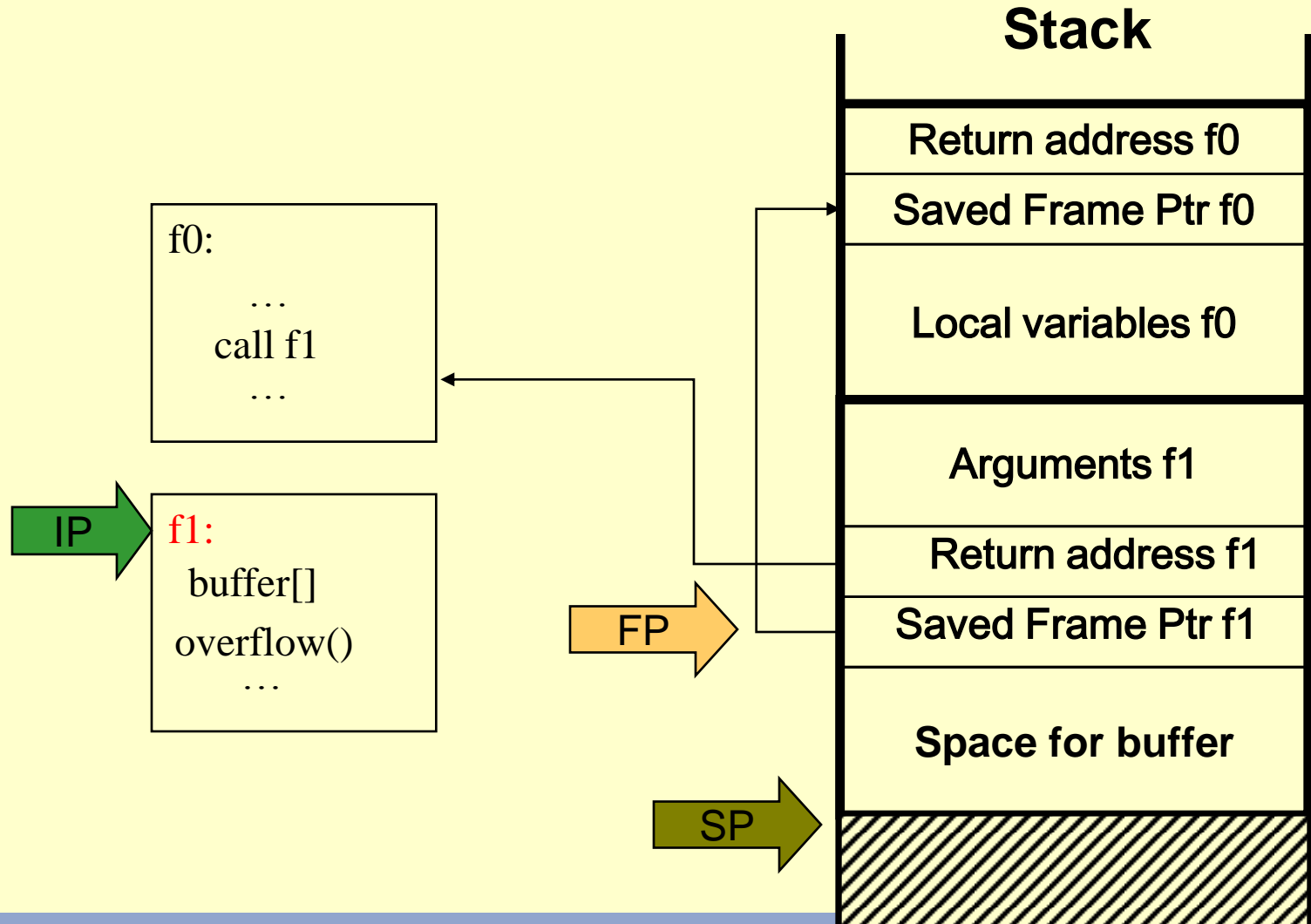
Stack based buffer overrun

- The stack is a memory area used at run time to track function calls and returns
 - Per call, an *activation record* or *stack frame* is pushed on the stack, containing:
 - Actual parameters, return address, automatically allocated local variables, ...
- As a consequence, if a local buffer variable can be overflowed, there are interesting memory locations to overwrite nearby

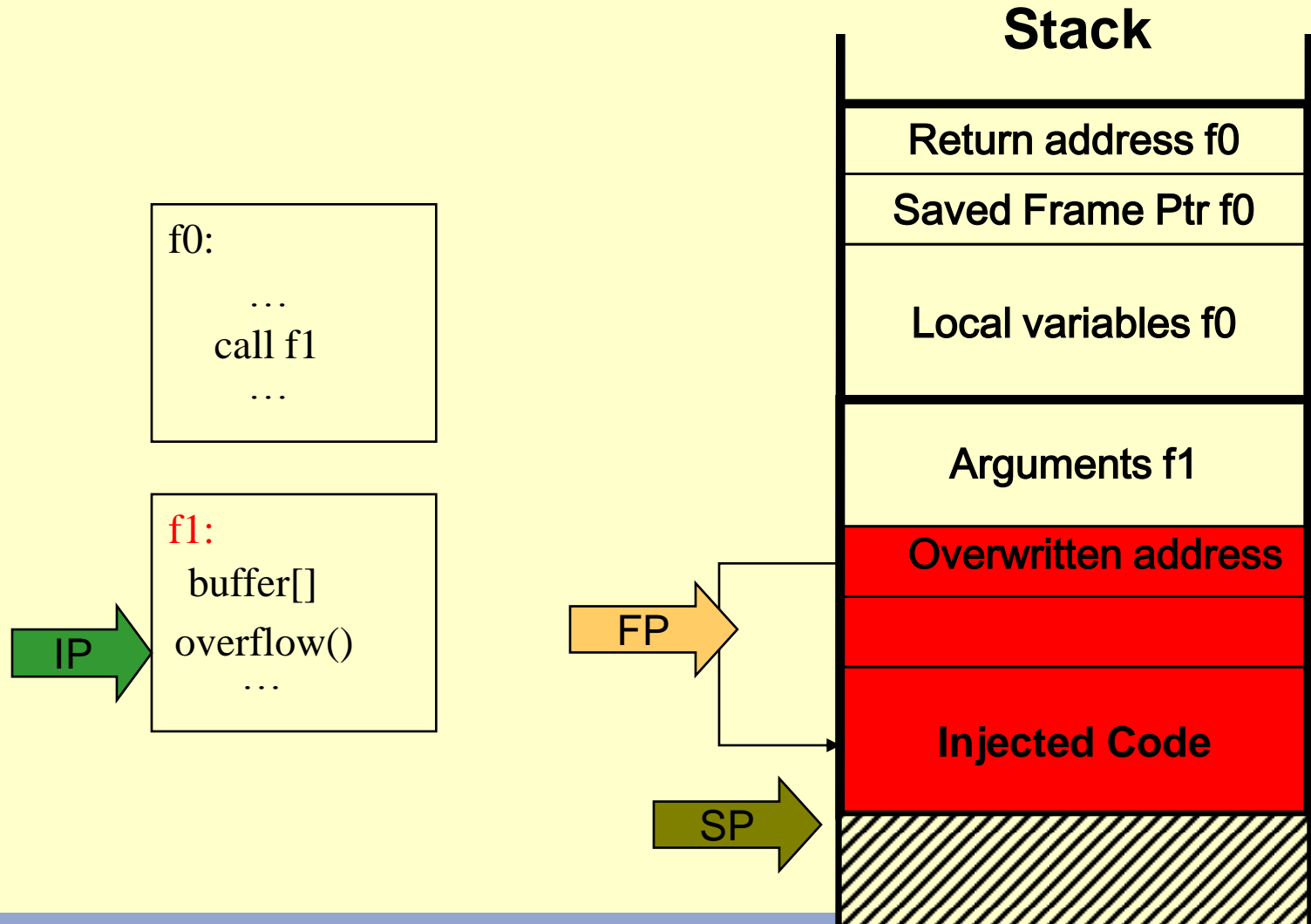
Stack based buffer overrun



Stack based buffer overrun



Stack based buffer overrun



Stack based buffer overrun

- Shell code strings:

LINUX on Intel:

```
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\x40xcd"  
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

SPARC Solaris:

```
char shellcode[] =
```

```
"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"  
"\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"  
"\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"  
"\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";
```


Stack based buffer overrun

- Example vulnerable program:

```
#include <string.h>
#include <stdio.h>

main(argc,argv)
int argc;
char **argv;
{ if (argc != 2) {
    printf("Usage: %s overflow\n",argv[0]);
    exit(1); }
  hole(argv[1]); }

hole(overflow)
char *overflow;
{ char buff[200];
  strcpy(buff ,overflow); }
```

Stack-based buffer overflow

- Lots of details to get right before it works:
 - No nulls in (character-)strings
 - Filling in the correct return address:
 - Fake return address must be precisely positioned
 - Attacker might not know the address of his own string
 - Other overwritten data must not be used before return from function
 - ...
- More information in
 - “Smashing the stack for fun and profit” by Aleph One

Overview

- Introduction
- Illustrating the risks of unsafe languages
- Safety and type soundness
- Sandboxing
- Conclusion

Safety

- A programming language is *safe* if its behavior is always well-defined
 - E.g. $a[i] = (\text{int}) x.f()$
- A safe language
 - Protects its own abstractions (e.g. no stack smashing attack)
 - Is inherently portable
- An unsafe language puts the burden of avoiding undefined situations on the programmer

Safety

- Some safe programming languages:
 - Java, C#, ML, Prolog, Scheme, Lisp
 - (Note: some of these languages have some unsafe features)
- Some generally unsafe programming languages:
 - C, C++, Pascal

(Although one could theoretically achieve safe instances of these languages by further defining the undefined behavior in a safe way)

Achieving safety

- By taking features out of the language
 - E.g. pointer arithmetic
- By means of run-time checking
 - E.g. array bounds checks
- By means of type checking
 - E.g. checking whether methods are defined appropriately
 - Requires type soundness

Types

- *Types* annotate program elements to assert certain invariant properties
 - E.g. This variable will always hold an integer
 - E.g. This variable will always refer to an object of class X (or one of its subclasses)
 - E.g. This array will never store more than 10 items
- *Type checking* verifies the assertions
- A language is *type sound* if the assertions are guaranteed to hold at run-time

```
using System;
```

```
public class Demo  
{
```

```
    static private string greeting = "Hello ";
```

```
    static void Main(string[] args)
```

```
    {
```

```
        foreach (string name in args)
```

```
        {
```

```
            Console.WriteLine(sayHello(name));
```

```
        }
```

```
    }
```

```
    static public string sayHello(string name)
```

```
    {
```

```
        return greeting + name;
```

```
    }
```

```
}
```

Field greeting is only accessible
by code in class Demo

Method sayHello() will always return
a value of type string.

Method sayHello() will always be
called with one parameter. That
parameter will be of type string.

Safe / Type sound languages

- Safe, typed and type sound languages:
 - Java, C#, ML
- Safe, untyped languages:
 - Lisp, Prolog, many interpreted languages
- Unsafe, typed languages
 - C, C++, Pascal
 - E.g. using pointer arithmetic in C, you can do anything you want and break any assertion made by the type system

Example

```
class DiskQuota {  
private:  
    long MinBytes;  
    long MaxBytes;  
};
```

```
void EvilCode(DiskQuota* pdq) {  
    // use pointer arithmetic to index  
    // into the object wherever we like!  
    ((long*)pdq)[1] = MAX_LONG;  
}
```

(Example taken from Keith Brown's April 2004 Security Brief)

Safety and type soundness for security

- Safety for security
 - Programs are invulnerable to attacks such as stack smashing attacks
- Typing for security
 - Can find bugs at compile time
- Type soundness for security
 - Can improve efficiency of safe languages
 - Can provide basic protection against untrusted code:
 - e.g. guarantee that untrusted code cannot access private fields of existing objects
 - Requires type checking at load time

Overview

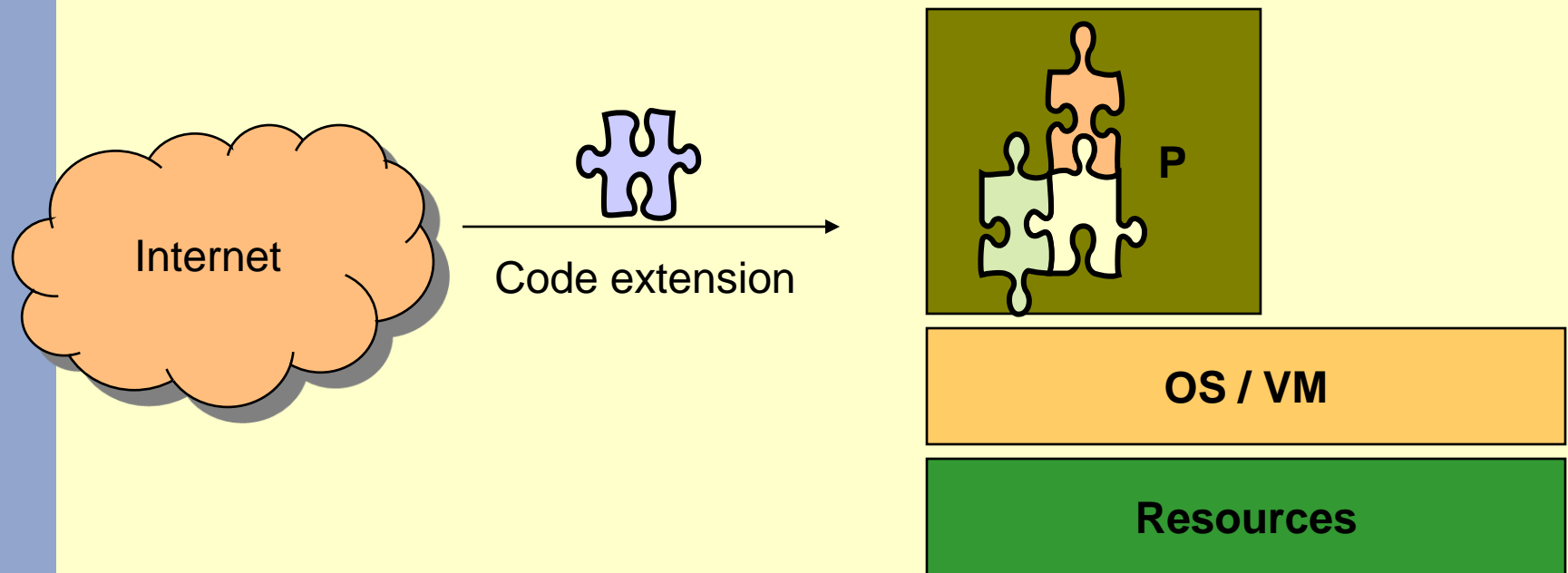
- Introduction
- Illustrating the risks of unsafe languages
- Safety and type soundness
- Sandboxing
- Conclusion

Extensible applications

- Modern applications often support run-time extensibility, possibly with downloaded code
 - Applets or controls on web pages
 - Browser plug-ins
 - Web server extensibility (JSP / ASP)
 - Multimedia codecs
 - ...

Extensible applications

- One OS process executes the application itself and all of its (possibly less trusted) extensions



Classic OS Access Control fails

- One session or subject (process) typically has a fixed set of permissions
- With untrusted code, the permissions of a subject may need to be reduced if the subject is currently executing less trusted code
- => Other access control architecture is needed

Terminology and concepts

- A *component* is a piece of software that is:
 - A unit of deployment
 - Third party composable
- An application can consist of multiple components
 - Some of these components are trusted more than others
- An application can be extended at runtime with new components
- We need security technologies that enables secure execution of such applications

Sandboxing: overview

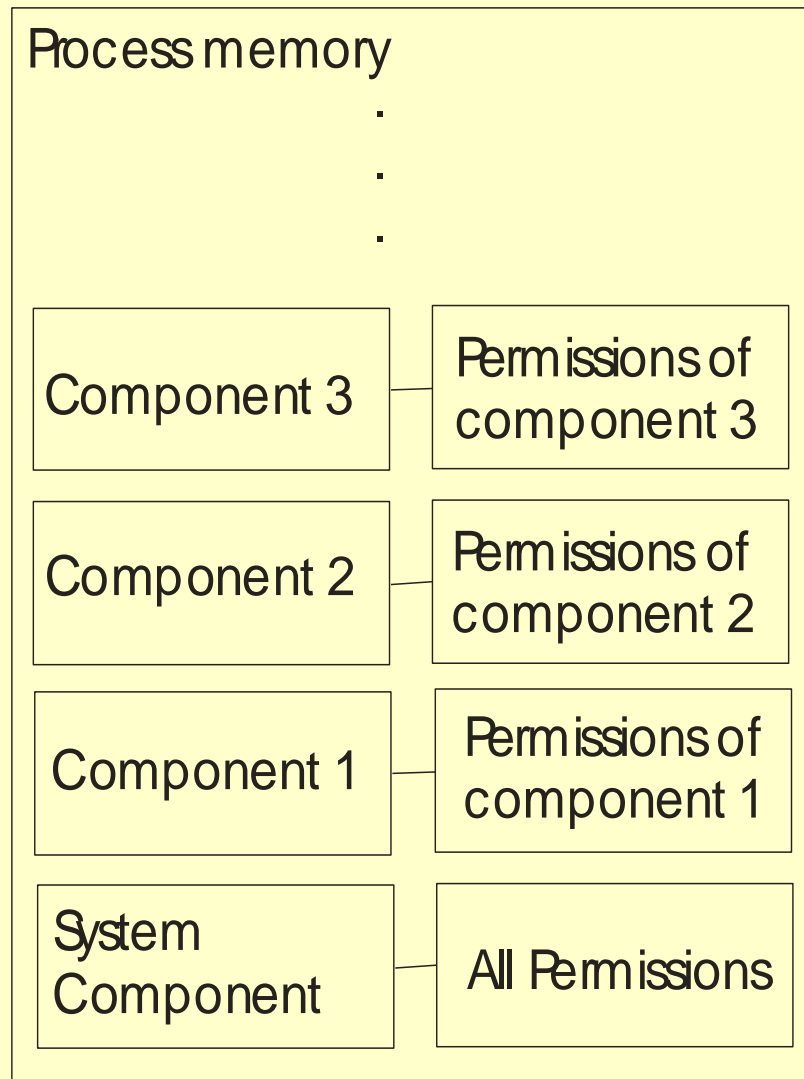
- *Permissions* encapsulate rights to access resources or perform operations
- A *security policy* assigns permissions to each component
- Every resource access or sensitive operation contains an explicit check that:
 - Through *stack inspection* finds out what components are active
 - Returns silently if all is OK, and throws an exception otherwise

Permissions

- Permission is a representation of a right to perform some actions
- Examples:
 - FilePermission(name, mode) (wildcards possible)
 - NetworkPermission
 - WindowPermission
- Permissions have a set semantics, hence one permission can imply (be a superset of) another one
 - E.g. FilePermission(“*”, “read”) implies FilePermission(“x”, “read”)
- Developers can define new custom permissions

Security Policy

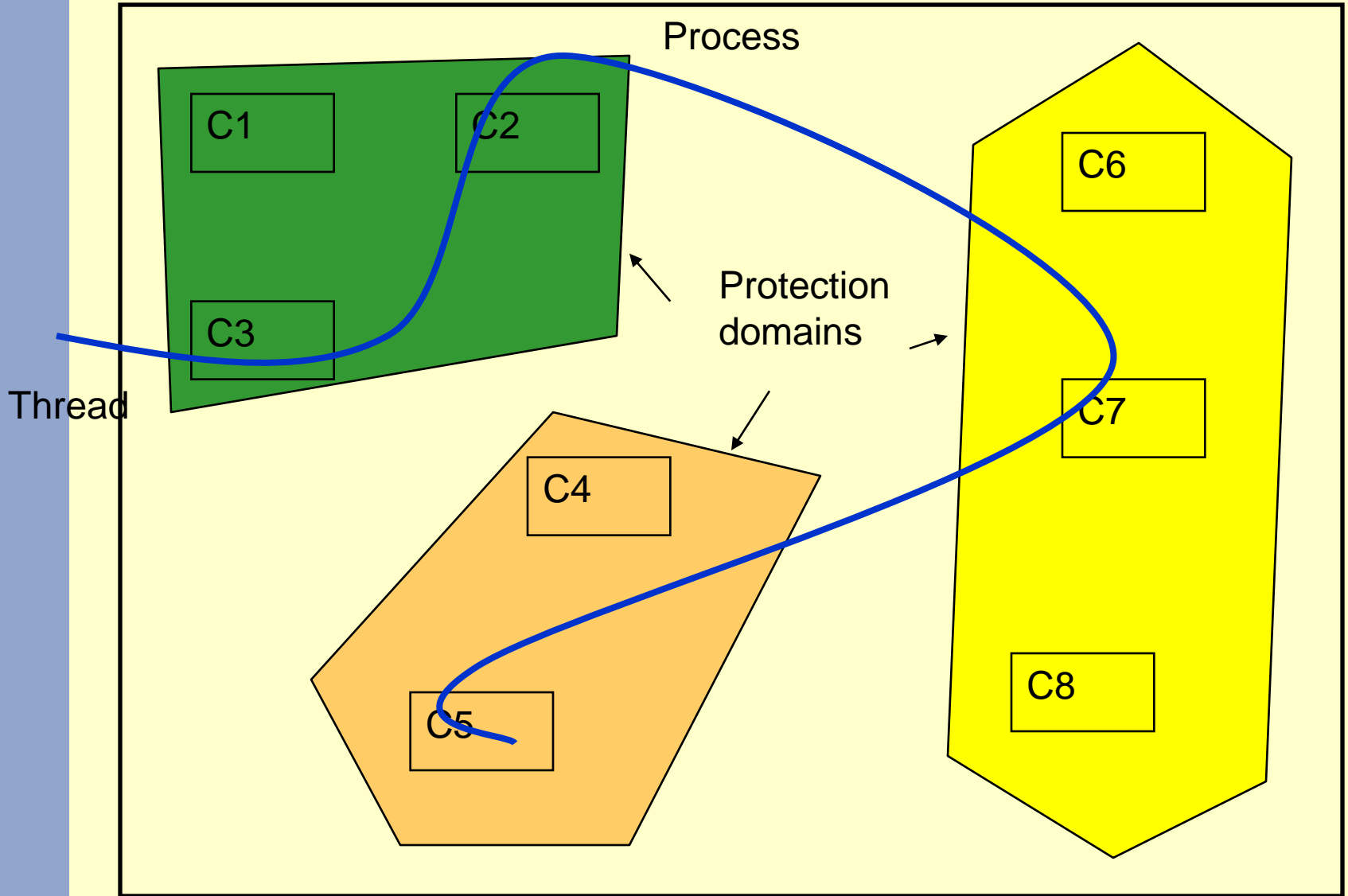
- A security policy assigns permissions to components
- Typically implemented as a configurable function that maps *evidence* to permissions
- Evidence is security-relevant information about the component:
 - Where did it come from?
 - Was it digitally signed and if so by whom?
- When loading a component, the VM consults the security policy and remembers the permissions



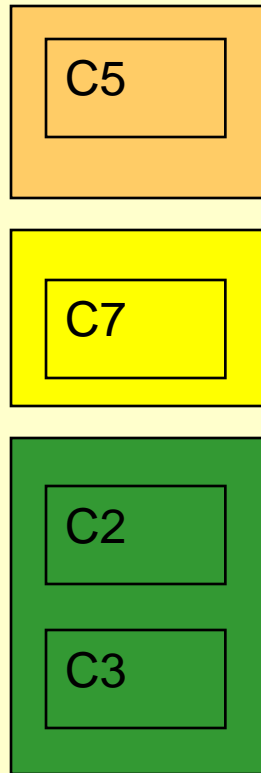
Components and their permissions in VM memory

Stack inspection

- Every resource access or sensitive operation exposed by the platform class library is protected by a `demandPermission(P)` call for an appropriate permission `P`
- The algorithm implemented by `demandPermission()` is based on *stack inspection* or *stack walking*



Stack walking: basic concepts



Stack for thread T

- Suppose thread T tries to access a resource
- Basic rule: this access is allowed if:
 - All components on the call stack have the right to access the resource

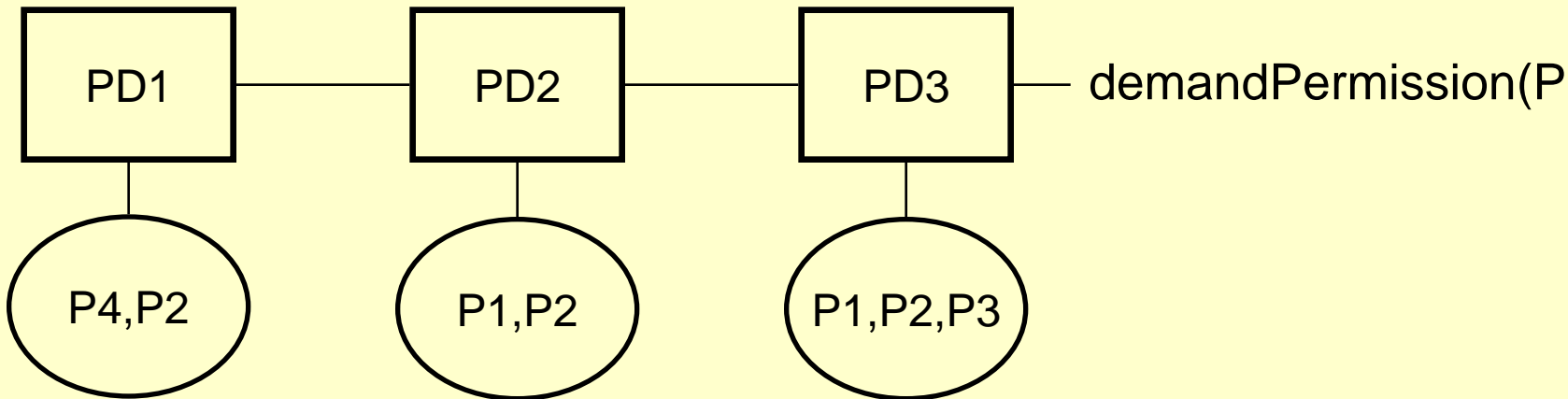
Stack walk modifiers

- Basic algorithm is too restrictive in some cases
- E.g. Giving a partially trusted component the right to open marked windows without giving it the right to open arbitrary windows
- Solution: stack walk modifiers

Stack walk modifiers

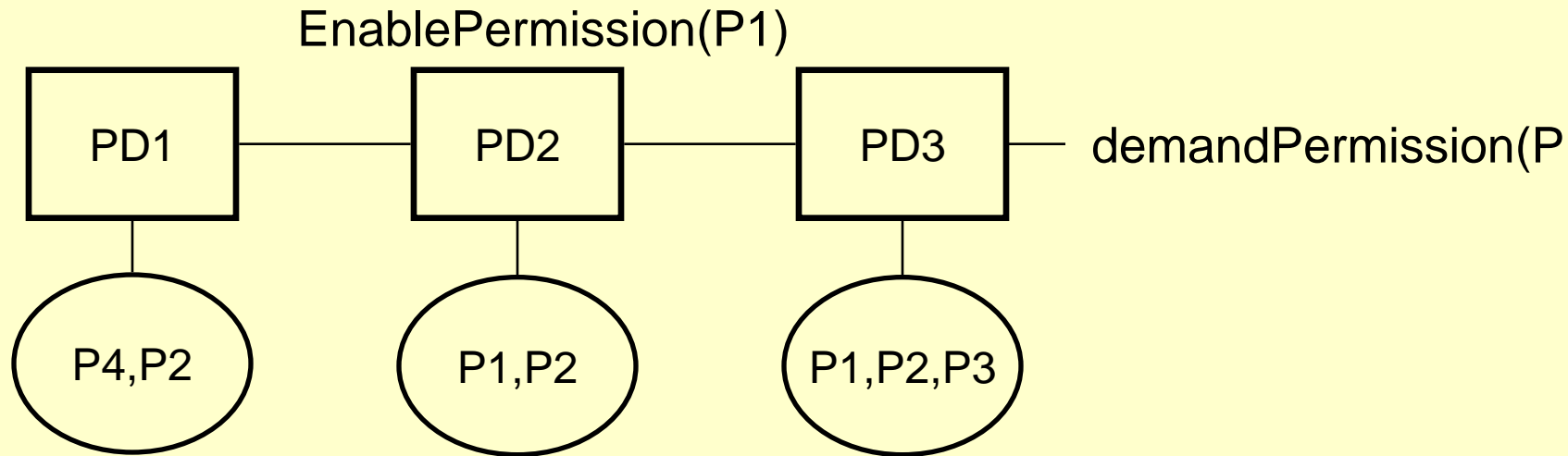
- `Enable_permission(P)`:
 - Means: don't check my callers for this permission, I take full responsibility
 - Essential to implement *controlled* access to resources for less trusted code
- `Disable_permission(P)`:
 - Means: don't grant me this permission, I don't need it
 - Supports principle of least privilege

Stack walk modifiers: examples



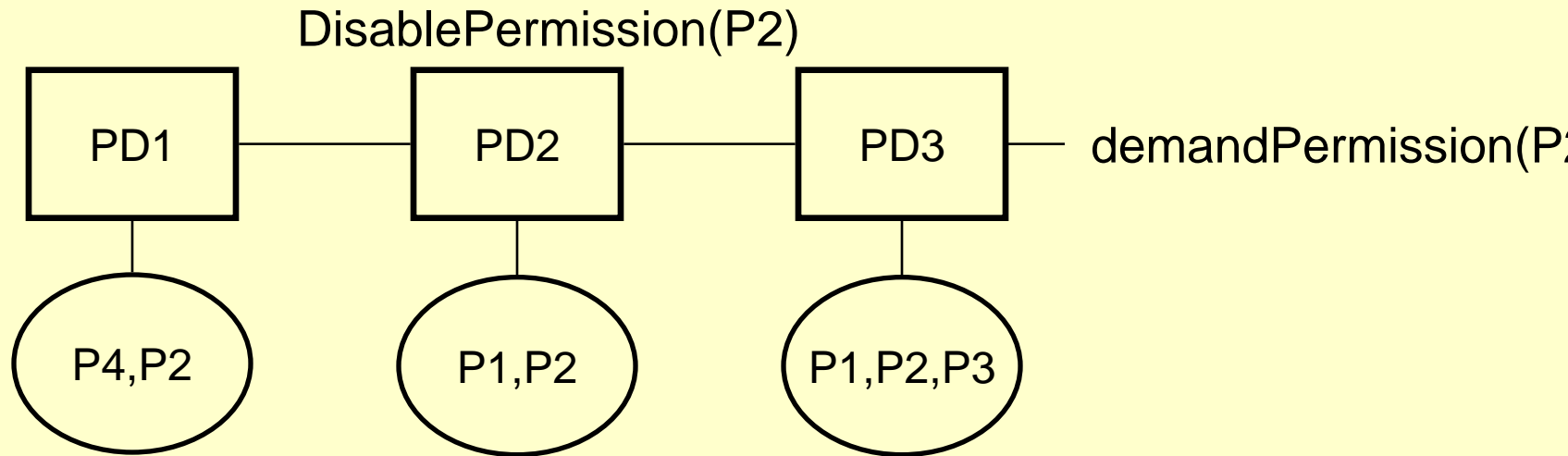
DemandPermission(P1) fails because PD1 does not have Permission P1

Stack walk modifiers: examples



DemandPermission(P1) succeeds

Stack walk modifiers: examples



DemandPermission(P2) fails

The applet window example

```
class Applet {  
    void showResults() {  
        Lib.openMarkedWindow();  
        ...  
    }  
}
```

```
class Lib {  
    void openMarkedWindow() {  
        // enable WindowPermission  
        openWindow();  
        // make sure this window  
        // is labelled  
    }  
}
```

openWindow()

openMarkedWindow()

showResults()

openWindow()

openMarkedWindow()

showResults()

enable
WindowPermission

(a) demandPermission fails

(b) demandPermission succeeds

Stack walking: algorithm

- On creation of a new thread: inherit access control context of creating thread
- DemandPermission(P) algorithm:

for each caller on the stack, from top to bottom:

if caller lacks Permission P:
throw exception

if caller has disabled Permission P:
throw exception

if caller has enabled Permission P:
return

check inherited access control context

Overview

- Introduction
- Illustrating the risks of unsafe languages
- Safety and type soundness
- Sandboxing
- Conclusion

Conclusion

- Programming language is one of the key tools of a developer
- It can have a serious impact on security:
 - Safety of the language
 - Support for sandboxing
 - *Support for compile-time detection of bugs*
 - *Well-designed API's for security*
- Further improvement of programming languages is an active research domain