Source code security

I. Source code security – Overview
II. Common mistakes:
signed-unsigned, integer over / under - flow...
I. Smashing the stack. Heap Overrun
II. Array Indexing Errors
III. File naming / working vulnerabilities
IV. Race condition. Multithread programming security.
V. String function library vulnerability
VI. DB access security - SQL Injection
VII. Protecting program code from reverse engineering. Obfuscators.
VIII. Virtual machine / environment vulnerabilities : .NET
IX. Denial of service (DoS)
X. Exam

Source code security - overview

A. Source code security myth
B. Most frequent causes:
   - New programming technologies, partial tested
   - Unprepared application to be run in an interconnected environment (World Wild Web !)
   - Input data quality; validation !!
   - Using unsafe function library, internally unsecure
   - Distance DB access insecurity
C. Language dependency: strong – unsafe ratio; see pointer vulnerability in C++

Source code security - overview

D. Why don't write secure / reliable code ?

E. Tradeoff:

- how much is robustness cost
- insecurity cost (stake, probability !)
- next improving cost

F. Machine dependency; difficult to build a context; circumstances compounding

G. Open source code ...

Evolution of a Security Exploit

1. Someone finds a bug.
2. Someone (usually the same) writes a demonstrative exploit.
3. Someone standardizes the exploit.
4. Script kiddies can now use the exploit to break in.
5. A worm can be written to automate exploit.

Source code security - overview

- Microsoft Clip Art Gallery had a **buffer overrun** issue causing random code running.
- File restoring function under Solaris (***ufrestore***) allowed to a anonymous user to become admin (root).
- Sort command in many like Unix systems, including OSX from Apple, allowed a **denial of service (DoS)** vulnerability.

Common mistakes: signed-unsigned

```
bool SafeCopy(char* str, int size)
 {
   char buf[80];
   if(size < sizeof(buf))
    {
      strcpy(buf, str); return true;
    }
     else return false;
 }
```

What return the following call:

```
printf( "\n %s", SafeCopy("Safe copy ???",-2) ?
                                        "Yes": "No!!" );
```

Common mistakes: integer overflow

```
bool AllocateStructs (void** ppMem ,
     unsigned short DimStruct, unsigned short NrElem)
{
   unsigned short bytes_necessary;
   bytes_necessary = DimStruct * NrElem;
  *ppMem = malloc(bytes_necessary);
   if(*ppMem == NULL) return false;
                    else return true;
}
```

```
unsigned short nrBytesAlloc = nr*sizeof(Product);
 printf("  Allocated: %u bytes instead of %u !!!",
                        nrBytesAlloc, 10000*sizeof(produs) );
```

Common mistakes: integer underflow

```
bool Allocate
     (void** ppMem, unsigned short NrBytes)
{
   NrBytes-=2; // could save 2 B ?!
  *ppMem = malloc(NrBytes );
   if(*ppMem == NULL) return false;
                                    else return true;
}
```

How many bytes allocates the call:

```
 char  *pZona; unsigned nrBytes=1;
Allocate((void**)&pZona, nrBytes );
```

Pointers and references

A. Declaring / using pointers
B. Memory leak; dinging pointer

```
void main()
{
    int *aux = new int,    **table = new int* [10];
    int i;
    for(i=0; i<10; i++)
     {
            *aux=i; table[i] = aux;
     }
    for(i=0; i<10; i++)
            cout << *table[i]<<"  ";
}
```

Pointers and references

Function pointers. Reconfiguring application at runtime – *late binding*

```
void func(const char* input)
{
   char buf[10];
   // with no parameters printf  display stack content !!!!
   printf("Stack: \n%p\n%p\n%p\n%p\n%p\n%p\n\n");
   strcpy(buf, input);     printf("%s\n", buf);
   printf("Stack:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
void hack(void)
          {     printf("Hacked program! \n");}
void main(int argc, char* argv[])
{
   printf("Address of func = %p\n", func);
   printf("Address of hack = %p\n", hack);
   func(argv[1]);
}
```

Automatic / Static

```
#include <stdio.h>

void main( )
{
 for( int i=1; i<4; i++)
   {
     static int j=1; int k=1;
     printf("\n i=%d j=%d k=%d", i,j,k);
     j++; k++;
   }
   getchar();
 }
```

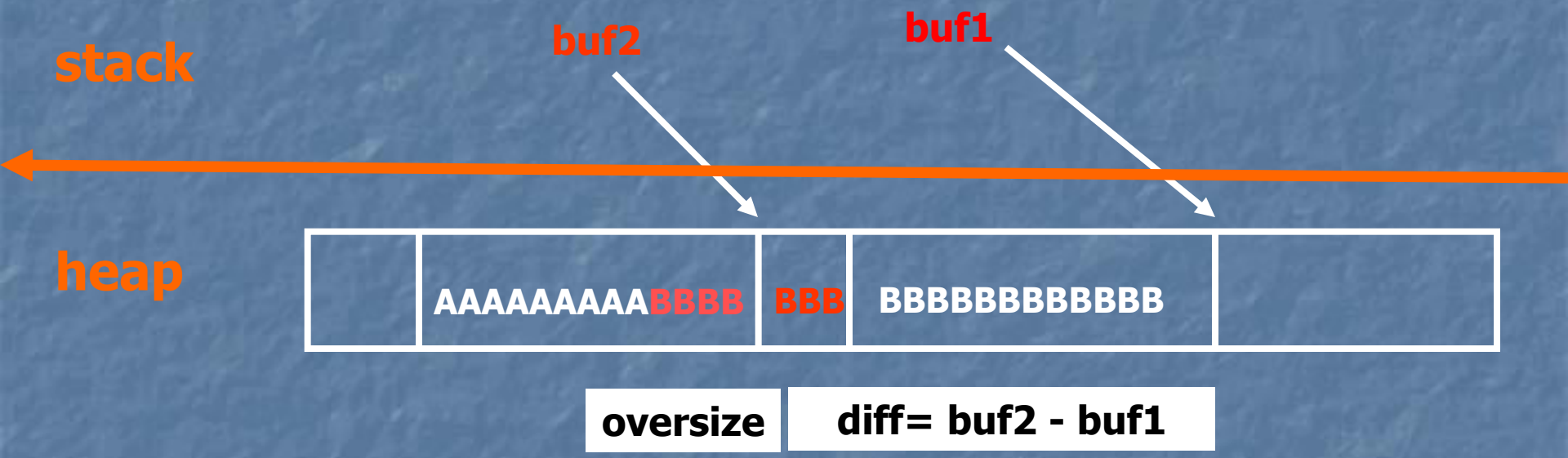
Buffer Overrun

- Write more data than the buffer can hold
- Example:
  - December 2008 attack on XML parser in Internet Explorer 7 - see http://isc.sans.org/diary.html?storyid=5458
- Two major types:
  - Heap overrun
  - Stack overrun

buffOver1.cpp

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

#define BUFSIZE 16
#define OVERSIZE 5
   /* nr de bytes din buf 2 ce vor fi suprascrisi */
void main()
{
  unsigned long diff;
  char * buf1 = (char *) malloc(BUFSIZE);
  char * buf2 = (char *) malloc(BUFSIZE);
  diff = (unsigned long )buf2 - (unsigned long)buf1;
  printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n",
             buf1, buf2, diff                    );

   memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';
   printf("before overflow: buf2 = %s\n", buf2);
   memset(buf1, 'B', (unsigned int )(diff + OVERSIZE));
   printf("after overflow: buf2 = %s\n", buf2); getchar();
}
```

Buffer overrun – vers. 1

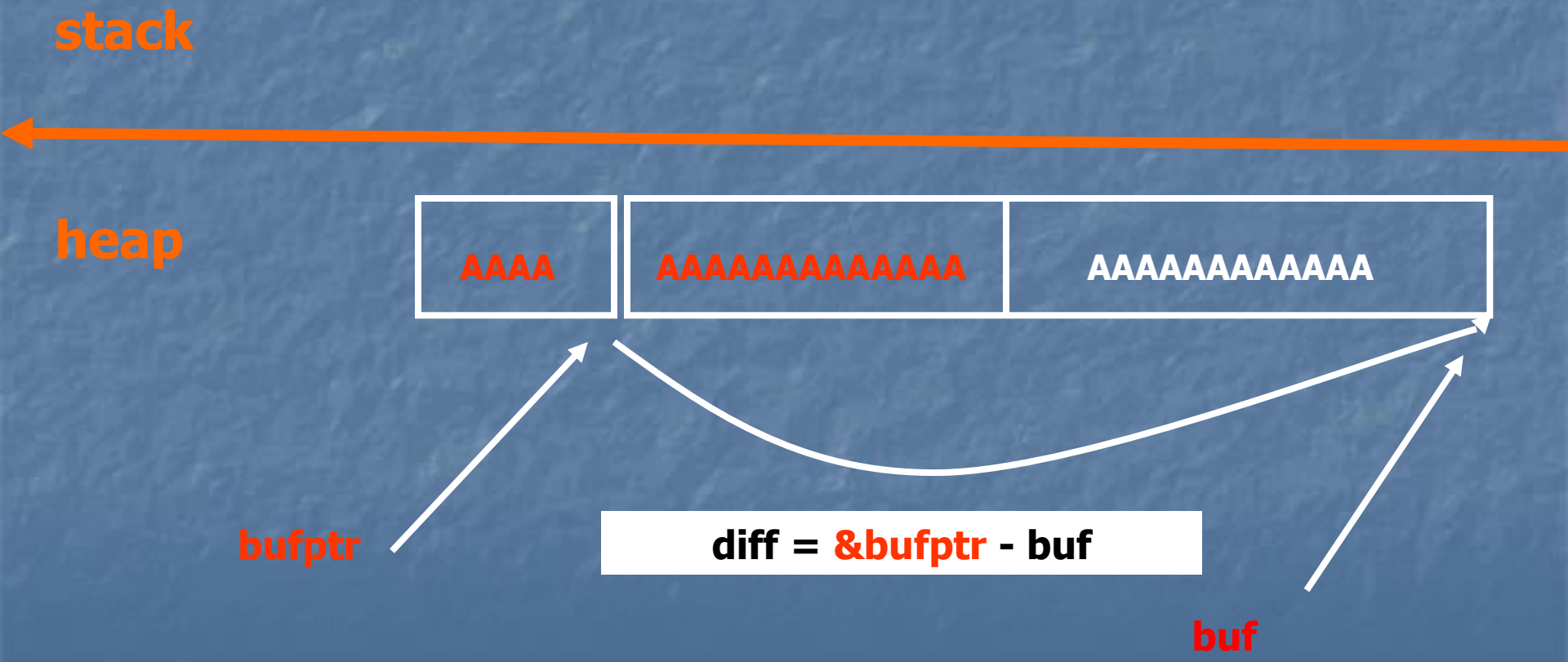


oversize	diff= buf2 - buf1

buffOver2.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#define BUFSIZE 16
#define ADDRLEN 4 /* address length */
void main()
{
  unsigned long diff;      static char buf[BUFSIZE] , *bufptr;

  bufptr = buf, diff = (unsigned long)&bufptr - (unsigned long)buf;
// pointer to bufptr, as static, resides in heap
  printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
       &bufptr, bufptr, buf, diff, diff);
   // initially, both bufptr and buf point the same area

  memset(buf, 'A', (unsigned int)(diff + ADDRLEN));
  printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
       &bufptr, bufptr, buf, diff, diff);   getchar();
// overwriting behind the area where the pointer is stored,
// the pointer will contain AAAA, as address !!!
}
```

Buffer overrun – vers. 2

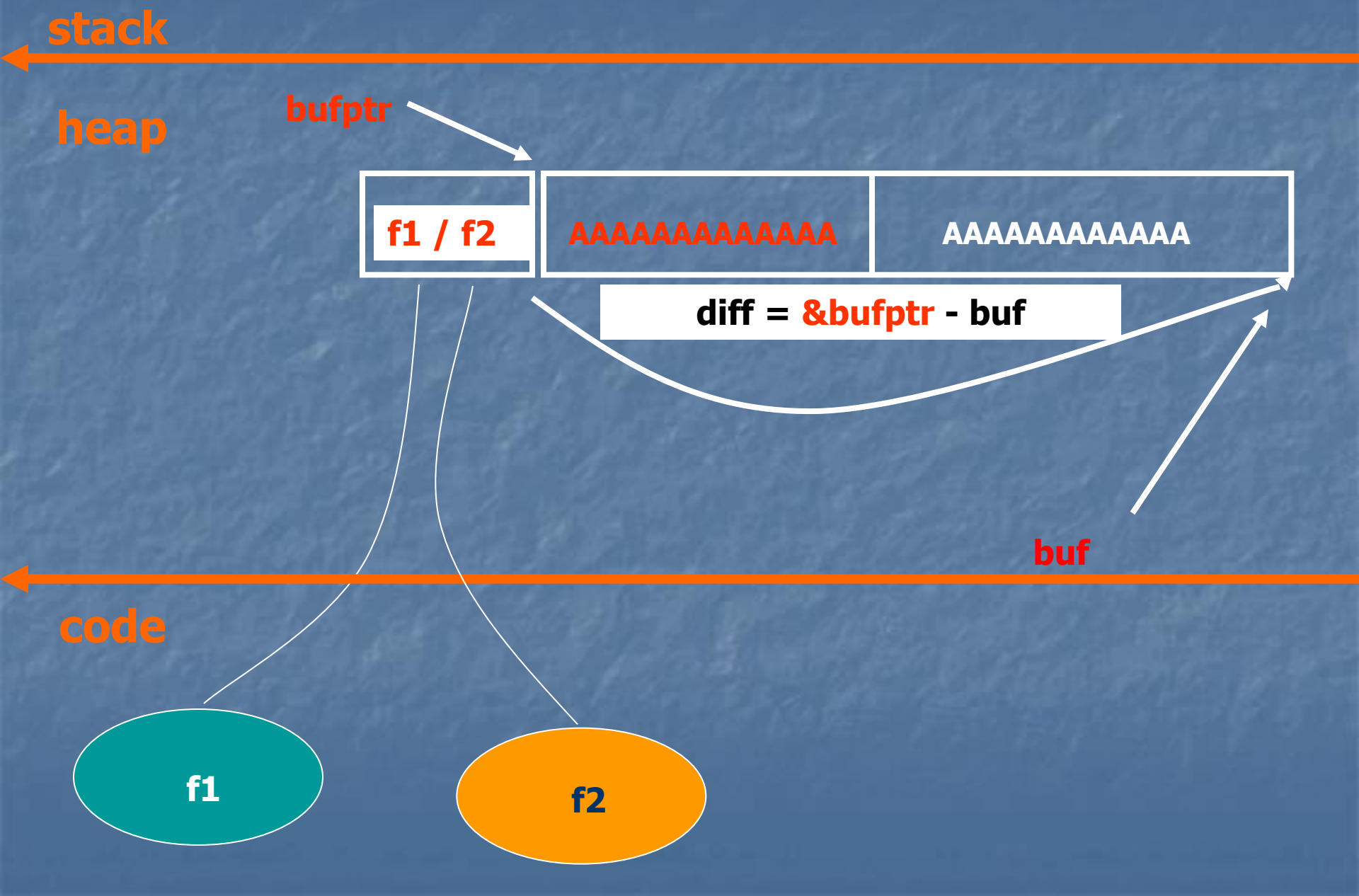


```
#include <stdio.h>

#include <string.h>

#define BUFSIZE 16
#define ADDRLEN 4 /* # of bytes in an address */

void f1()
    {                   printf("\nGood");      }

void f2()
    {                   printf("\nBad");       }
```

buffOver3.cpp (2)

```
void main()
  {
 unsigned long diff;
 static char buf[BUFSIZE];   static void(*bufptr)();
 bufptr = f1; // well intented
 diff = (unsigned long)&bufptr - (unsigned long)buf;
 printf("bufptr(%p)=%p,buf=%p,diff=0x%x (%d) bytes\n",
         &bufptr, bufptr, buf, diff, diff);
 bufptr(); // right function call !
 buf[diff]=0x41;            buf[diff+1]=0x10;
 buf[diff+2]=0x41;          buf[diff+3]=0x00;
 bufptr(); // diverted call !
 printf("\nf2=---%p----\n", f2);
 printf("bufptr (%p)= %p,buf=%p,diff=0x%x (%d)bytes\n",
         &bufptr, bufptr, buf, diff, diff);
getchar();
}
```

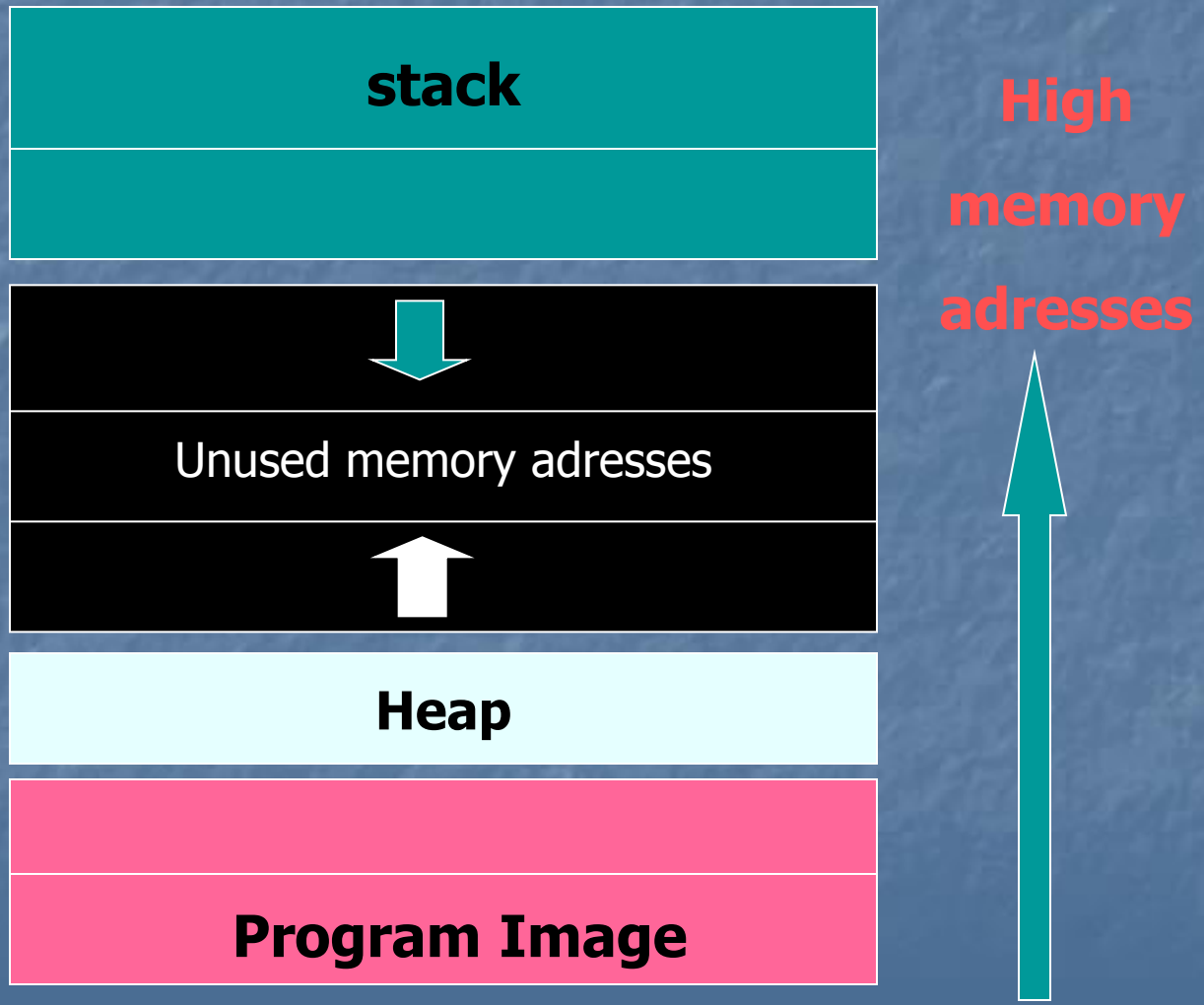
Buffer overrun – vers. 3



Stack Overrun

- The single most exploited vulnerability
- The first worm, called the "Morris Worm", used a stack overrun in "Sendmail" – 1988

Stack / Heap


stack
Unused memory adresses
Heap
Program Image
High
memory
adresses

Stack Overrun

- A few things to understand:
  - The stack usually grows downwards.
  - The stack frame in “C”
    - function arguments
    - return address
    - base pointer
    - local automatic variables

Stack walking

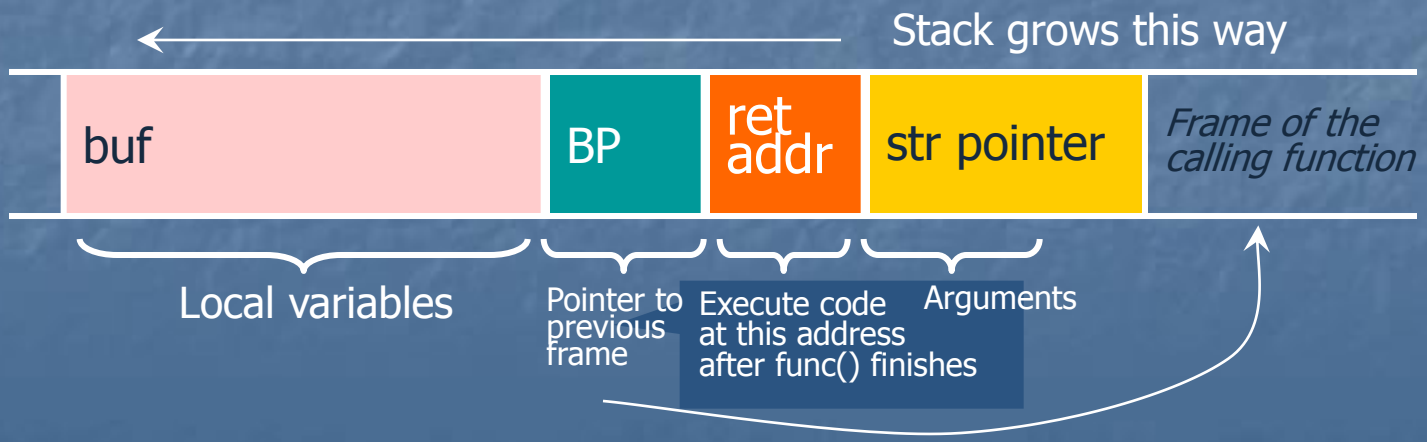
```
public static int f1(int x) { int r  = f2(x); return r + 1; }

public static int f2(int value) { int retval = value; return retval; }

main()  { f1(1); }
```

Stack walking

- value pointed by str is copied onto the stack…

```
void func(char *str)
{
        char buf[100];
        strcpy(buf,str);
}
```

- If a string longer than 99 bytes is copied into buffer, it will overwrite adjacent stack locations



Stack walking

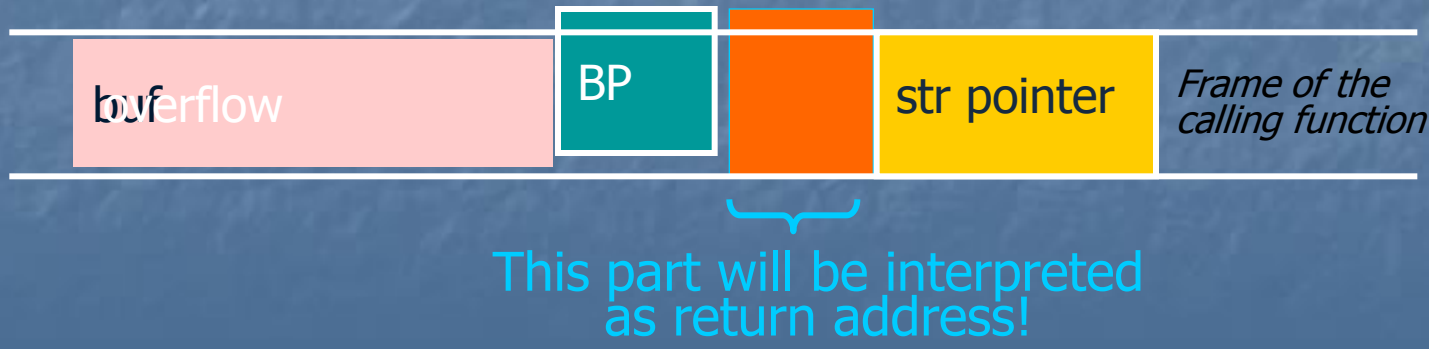
- value pointed by str is copied onto stack...

```
void func(char *str)
{
        char buf[100];

        strcpy(buf,str);
}
```

Allocate automatic buffer (only 100 B reserved on stack)

Copy without capacity checking

- If a string longer than 99 bytes is copied into buffer, it will overwrite adjacent stack locations



Stack

phrack3.cpp

```
void function(int a, int b, int c)
{
   char buffer1[5]="ABCD";
   char buffer2[10]="123456789";
   char *ret;

   ret =  buffer1 + 20; // point where return adress is located
  (*ret) += 8;
// modify to back in main, further (avoiding assignment)
  buffer1[0]='A';
}

void main()
{
  int x;   x = 0;   function(1,2,3);
           x = 1;
  printf("%d\n",x); getchar();
}
```

Stack

phrack4.cpp

```
void function(char *param1, int b, int c)
{
  char buffer1[5]="ABCD";    char buffer2[10]="123456789";
  char *ret;    ret =  buffer1 + 20; // address of return address
  strncpy(ret,param1,4);//put f2 as return address !
          buffer1[0]='a'; // used as breakpoint
}

void f2()  { printf("\nHack"); getchar(); exit(1);  }

void main()
{
 int x=0;  char s[4]={0xdb,0x11,0x41,0x00}; // f2 address
 printf("Address  of f2:%p\n",f2); // to know it later ...
 function(s,2,3); // go to bring the address of f2;
 x = 1;
 printf("%d\n",x); getchar();
}
```

Stack / Heap

testDebugger.cpp

```
#include <stdio.h>
int f( char *s1,char * s2)
{    char text[21];
     text[0]='A'; text[1]='B'; text[2]='C';text[3]='D';
     text[16]='P';text[17]='Q'; text[18]='R';
      text[19]='S';text[20]='T';
     // char text[21]="ABCDEFGHIJKLMNOPQRST";
     int y=5;
     return y;
}
void main( )
{
   int x;
  char *a="AAAAAAAAAAAAAAAAAAAAAA",
      *b="BBBBBBBBBBBBBBBBBB";
  f(a,b);
  x=1; getchar();
 }
```

arrayIndex.cpp

```
int* IntVector; // global pointer

void Attack(void)
{
      printf("\nAttention! Hack...\n");
      getchar(); // look at ...
      exit(1); // no traces
}


void WriteAt(unsigned long index, unsigned long whatToWrite)
{
      char buffer1[5]="ABCD";
      printf("\nWrite memory at : %p with %p",
                                    &(IntVector[index]), whatToWrite);

                  IntVector[index] = whatToWrite;
                  buffer1[0]='A';
}
```

```
bool AllocVector(int nElements)
{
   IntVector = (int*)malloc(  sizeof(int)* nElements  );
      printf("Address of IntVector: %d\n", IntVector);
   if(IntVector == NULL)        return false;
                  else        return true;
}
void  main(int argc, char* argv[])
{
   unsigned long index=0, adrAttack=0;
   int stack = 0x0012FE6C+4; printf("Address of return address: %p\n", stack);
   printf("Address of Attack function: %p ( decimal %d )\n", Attack, Attack);

   //Allocate a vector; 64 KB are enough ...
   if(!AllocVector(0xffff))
         {
            printf("Allocation failure !\n");
         }

      index = (stack + 0x100000000 -(int)IntVector ) /4;
      adrAttack = (int)Attack;
      // intialize as decimal, avoiding litle endian

      WriteAt(index, adrAttack); // hit far from ...
      getchar();;
 }
```

arrayIndex.cpp

Preventing execution of untrusted code

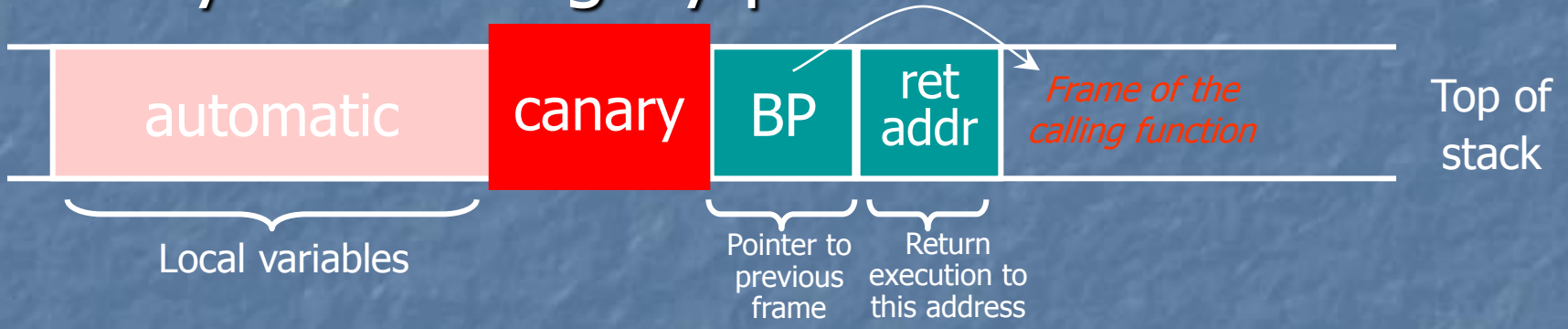
- Digitally signed all code
- Ensure that all control transfers are into a trusted, approved code image
- Mark all writeable memory locations as non-executable
  - Example: Microsoft's Data Execution Prevention (DEP); this will block (almost) all code injection exploits

Preventing Buffer Overrun

- Use safe programming languages (Java ?!)
  - What about legacy C code?
- Mark stack as non-executable (marking stack segment as non-executable exist for Linux, Solaris, OpenBSD. But LISP interpreters? Variable format compilers?)
- Randomize stack location
- Encrypt return address on stack by XOR-ing with random string; decrypt failure
- Run-time checking of array and buffer bounds
  - StackGuard, PointGuard, LibSafe,

Run-Time Checking: StackGuard

- Insert random value canary string in stack frames
- Any overflow will damage the canary too
- Verify their integrity prior to return in caller



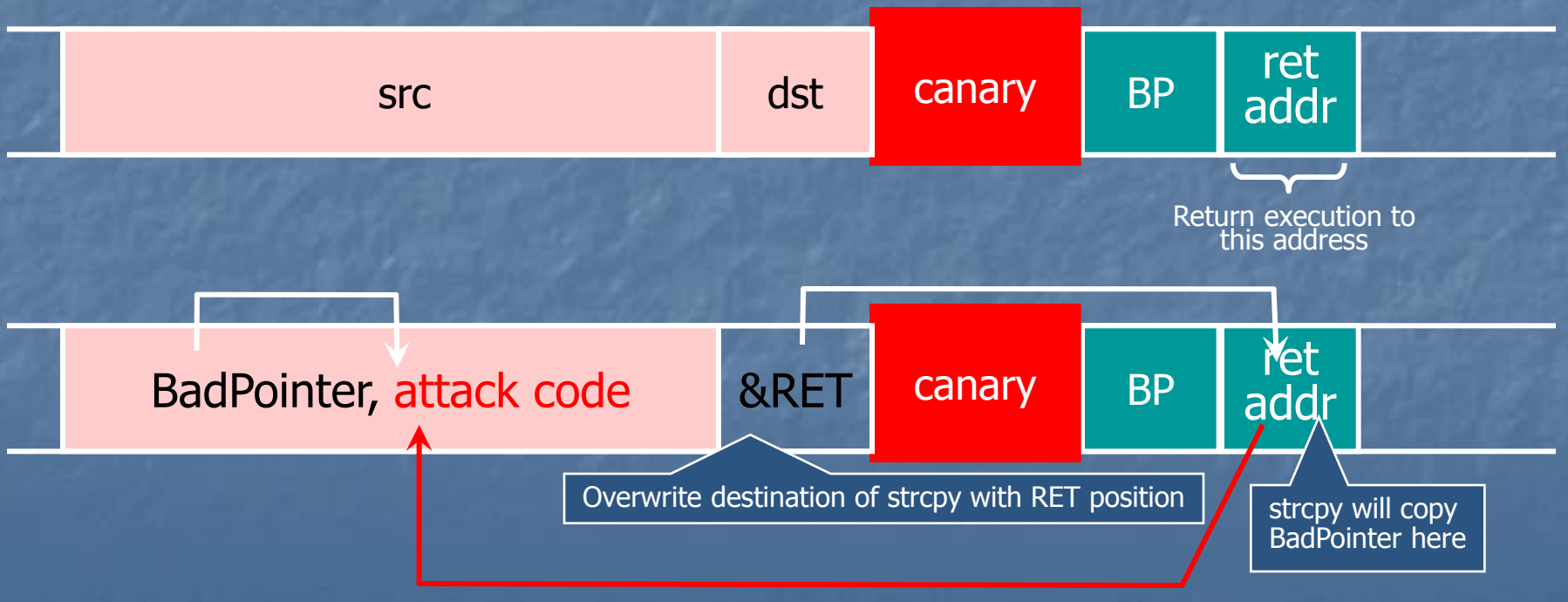
- Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF, string functions like strcpy won't copy beyond "\0" !

Run-Time Checking: StackGuard

- 8% performance penalty for Apache Web server
- StackGuard requires code recompiling
- StackGuard can be defeated!
  - Phrack Magazine article (Bulba and Kil3r)

Defeating StackGuard

- Suppose program contains strcpy(dst,srs) where attacker controls both dst and src
  - Example: dst is a local pointer variable



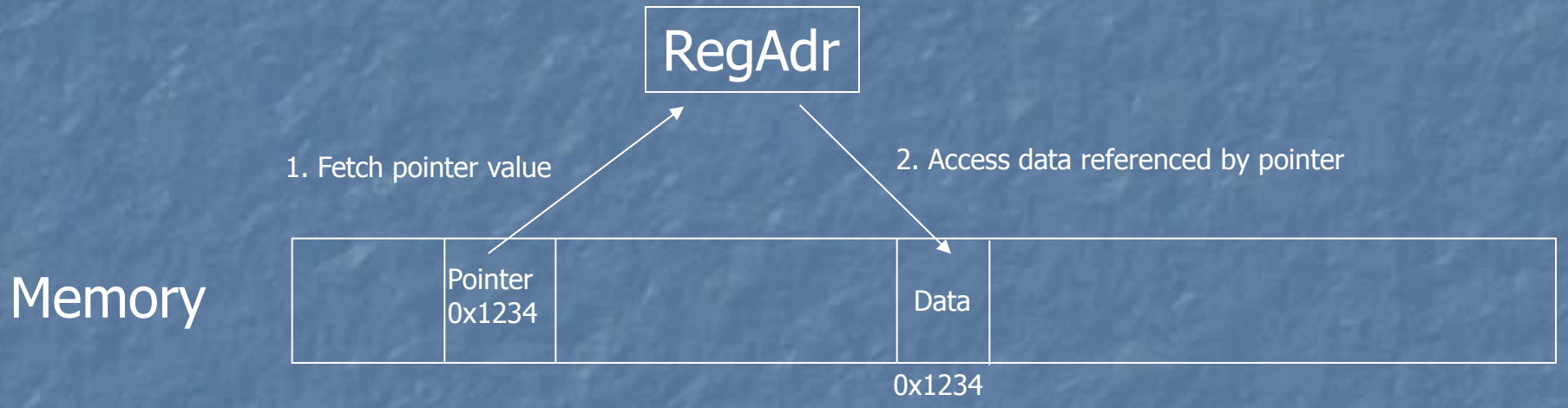
PointGuard

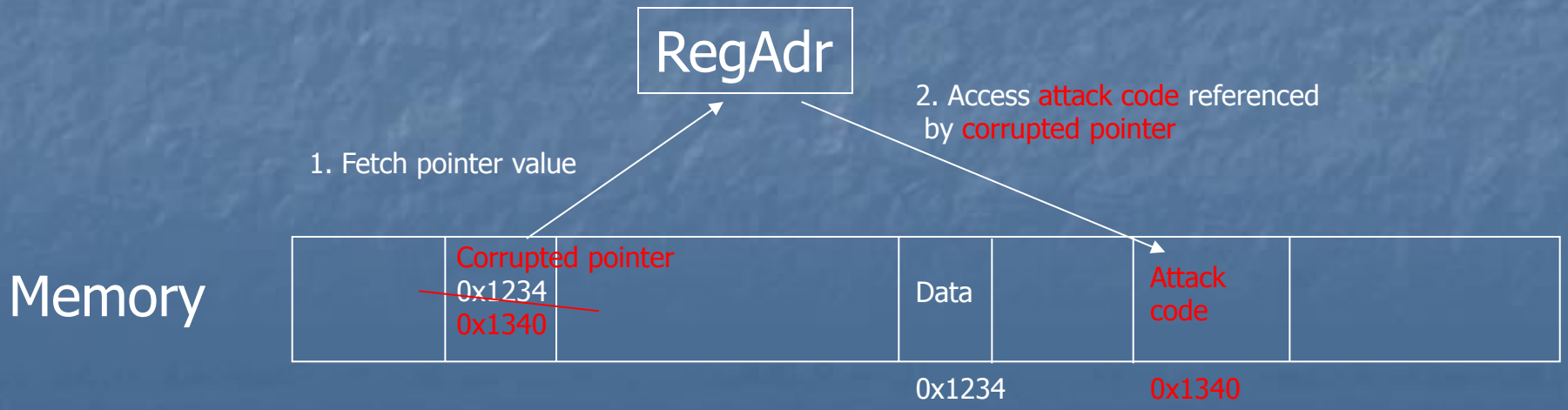
Encrypt all pointers while in memory

- Generate a random key at execution time
- Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
- Premise: pointers cannot be overflown while in registers

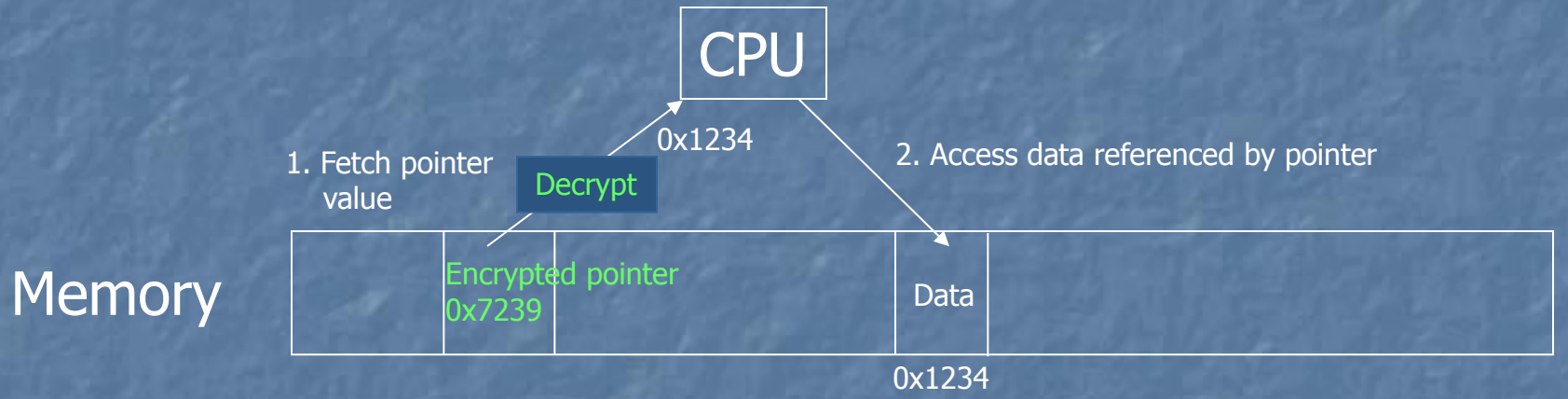
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

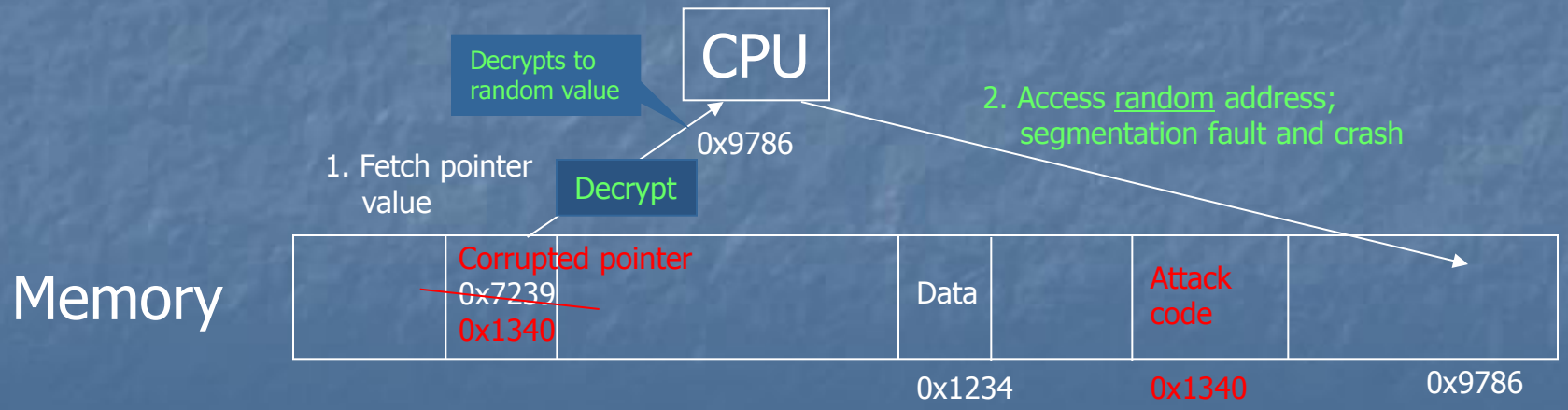
Normal Pointer Dereference [Cowan]





PointGuard Dereference [Cowan]





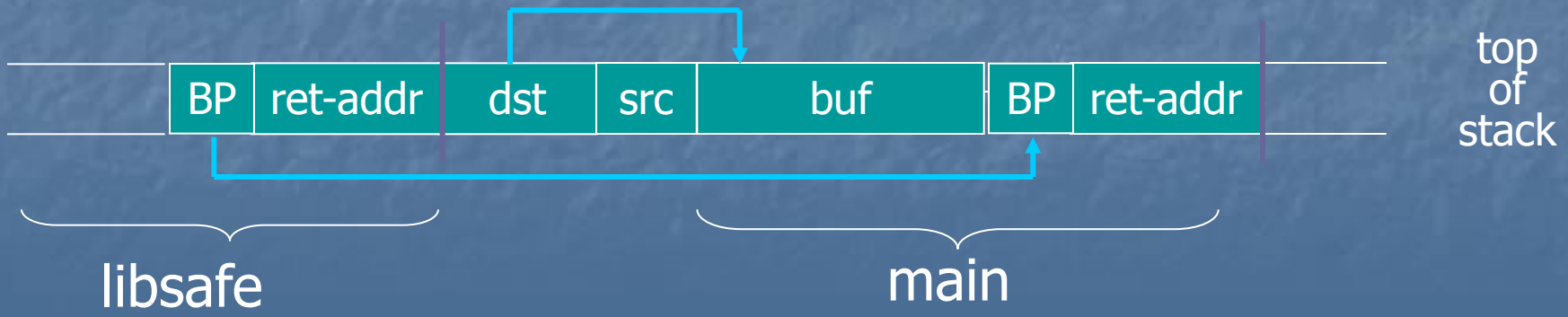
PointGuard Issues

- **Performance penalty** (pointer dereferences are very frequent)
- Compiler issues
  - Must encrypt and decrypt **only** pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- Point Guard-ed code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

Run-Time Checking: LibSafe

- Dynamically loaded library
- Intercepts unsafe calls /strcpy(dst,src)/
- Checks if there is sufficient space in current stack frame

  If | frame-pointer – dest | > strlen(src)
  - does strcpy; else terminates application



Heap Overrun

- Heap Overruns – Until 2002
  - Analyze the heap and search for convenient pointers
  - Exploit code highly dependant on exact program state (frame structure)

Heap Overrun

- Heap Overruns – after 2002
- Management of the individual allocation blocks is done with a data structure.
  - The pointers for that data structure are maintained in the same area as the heap
  - Writing past the end of a buffer change this structure.
  - When an application frees memory, free heap sections are merged
  - an attacker can cause arbitrary values to be written to arbitrary locations!

Race Condition

- a programming error in a multitask system when the system's work depends on the order in which code sections are executed
- several threads of a multithread application try to simultaneously get access to the data, meanwhile at least one thread performs saving.
- See THERAC-25 ACCIDENTS at http://www.cas.mcmaster.ca/%7Ecs3se3/htm/therac.htm

Race Condition

- a **process** has its own memory space, its own set of variables, its own stack and heap
- processes are an architectural construct.
- One process could spawn another process, but afterwards the two processes become more or less independently

Race Condition

- all **threads** within a process share the same state and same memory space, and can communicate with each other directly
- a **thread** is a coding construct
- a separate stream of execution that takes place simultaneously with and independently of everything else that might be happening

Race Condition

- **cooperative threading** that require a thread to explicitly yield control before other
- **preemptive threading** in which the virtual machine guarantees that all threads of the same priority get time in which to run

53_RaceConditions_Thread

55_ControllingRaceCondition

raceCondition.cs

```
public void Process()
   {
      Thread worker1 = new Thread(Work1);
      Thread worker2 = new Thread(Work2);
      worker1.Start();       worker2.Start();
      Thread.Sleep(2500);    Console.WriteLine(result);
   }
   void Work1()
   {
      int r = rndm.Next(2000);
      Thread.Sleep(r); Console.WriteLine("One " +r);
      result = 1;
   }
   void Work2()
   {
      int r = rndm.Next(2000);
      Thread.Sleep(r); Console.WriteLine("Two " +r);
      result = 2;
   }
```

Common mistakes: naming problem

Multiple ways of naming lots of things

Canonicalization – Conversion to a single, "standard" name

Files

- **/x/data or /y/z/../../x/data**
- **/y/z/%2e%2e/x/data**
- **links**

Others (DNS names, IP addresses, etc.)

Common mistakes: file naming problem

8.3 standard: FAT32 generate a name in 8.3format for long named file (Windows 16 bits need access this kind of file too

Calcul02Buget.xls ----------→ CALCUL~1.XLS

Calcul03Buget.xls ----------→ CALCUL~2.XLS

```
private static string[] FisiereSensibile =
        {       "Calcul02Buget.xls", "PlanFinanciar.Doc"   };

private static bool permiteAcces(string sir, string[] lista)
{
        foreach (string s in lista)
           if (s == sir) return false;
        return true;
 }
```

Common mistakes: file naming problem

Processing file name before accesing file:

- eliminate blanks and "\"
- eliminate "." when repeatly apears in file name
- file names are insensitive case

How can we exploit ?

Common mistakes: Temporary file manipulation

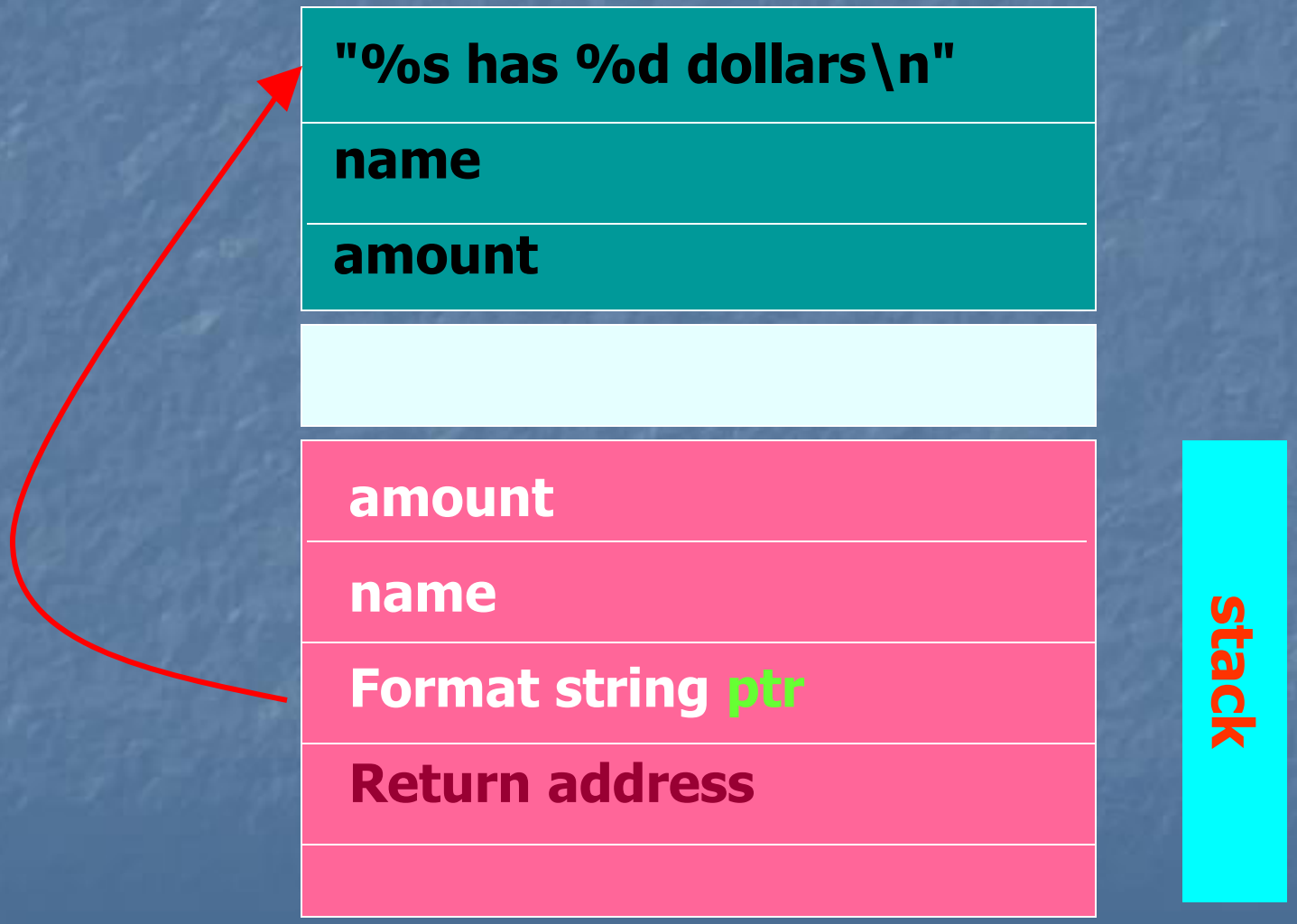
- Most common case - creating a temporary file in a known location and accesible to an attacker.
- Allows bypassing of the security model, changing internal program data structures and, in some cases, arbitrary code execution

Format Strings vulnerability

- Passing untrusted buffer as the format string in a *printf* call
- Easy to find during an audit, easy to fix.
- Easy to find in the binary, easy to exploit.

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

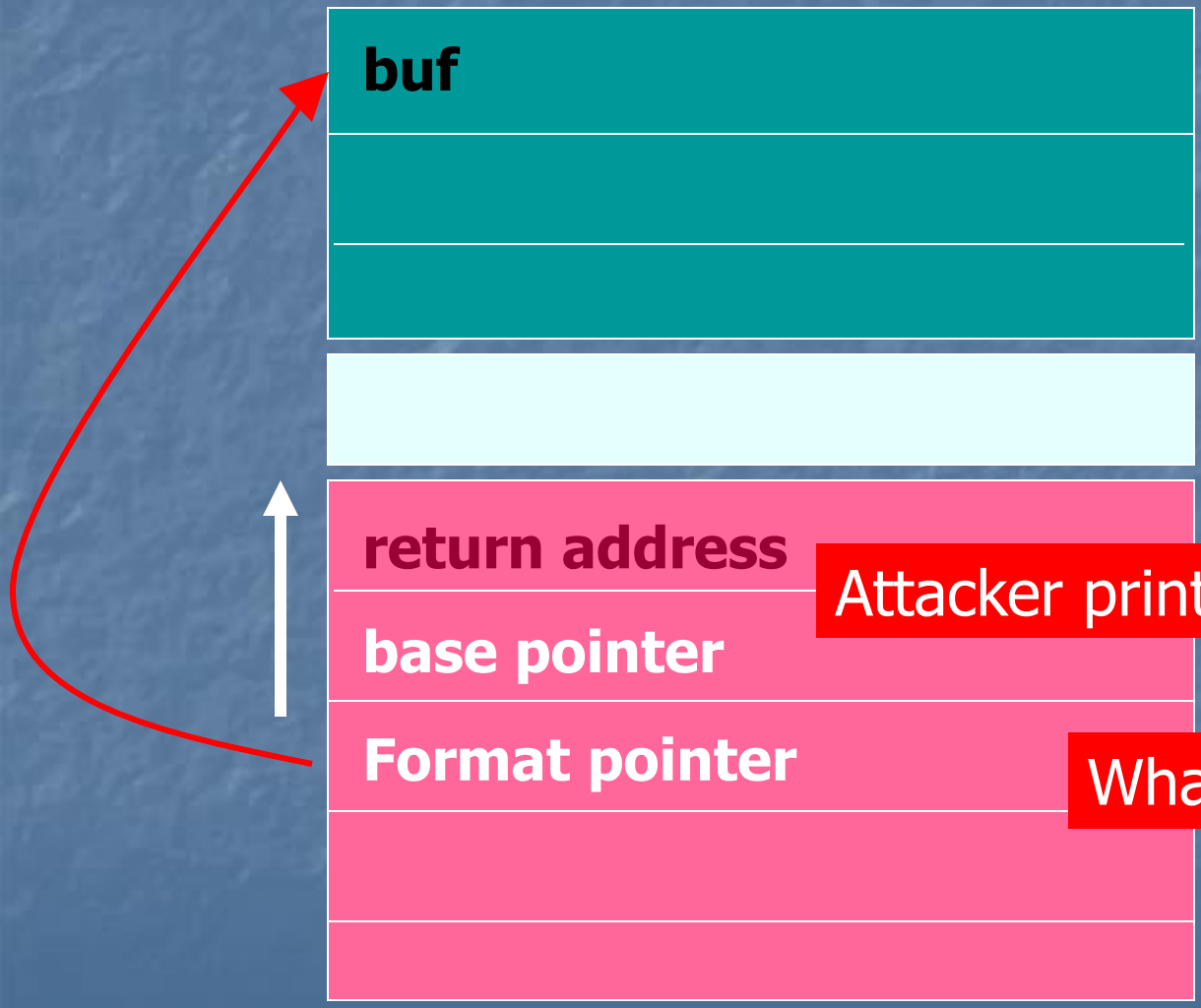
printf("%s has %d dollars\n", name, amount);



Format Strings vulnerability

- It is up to the programmer to make sure the parameters number and type match the format string descriptors.
- Some compilers will verify this for static format strings; (dynamic ?)
- A format string vulnerability that echoes the result to the attacker is like giving the attacker a debugger into the application.
- Arbitrary code execution exploitation is relatively easy

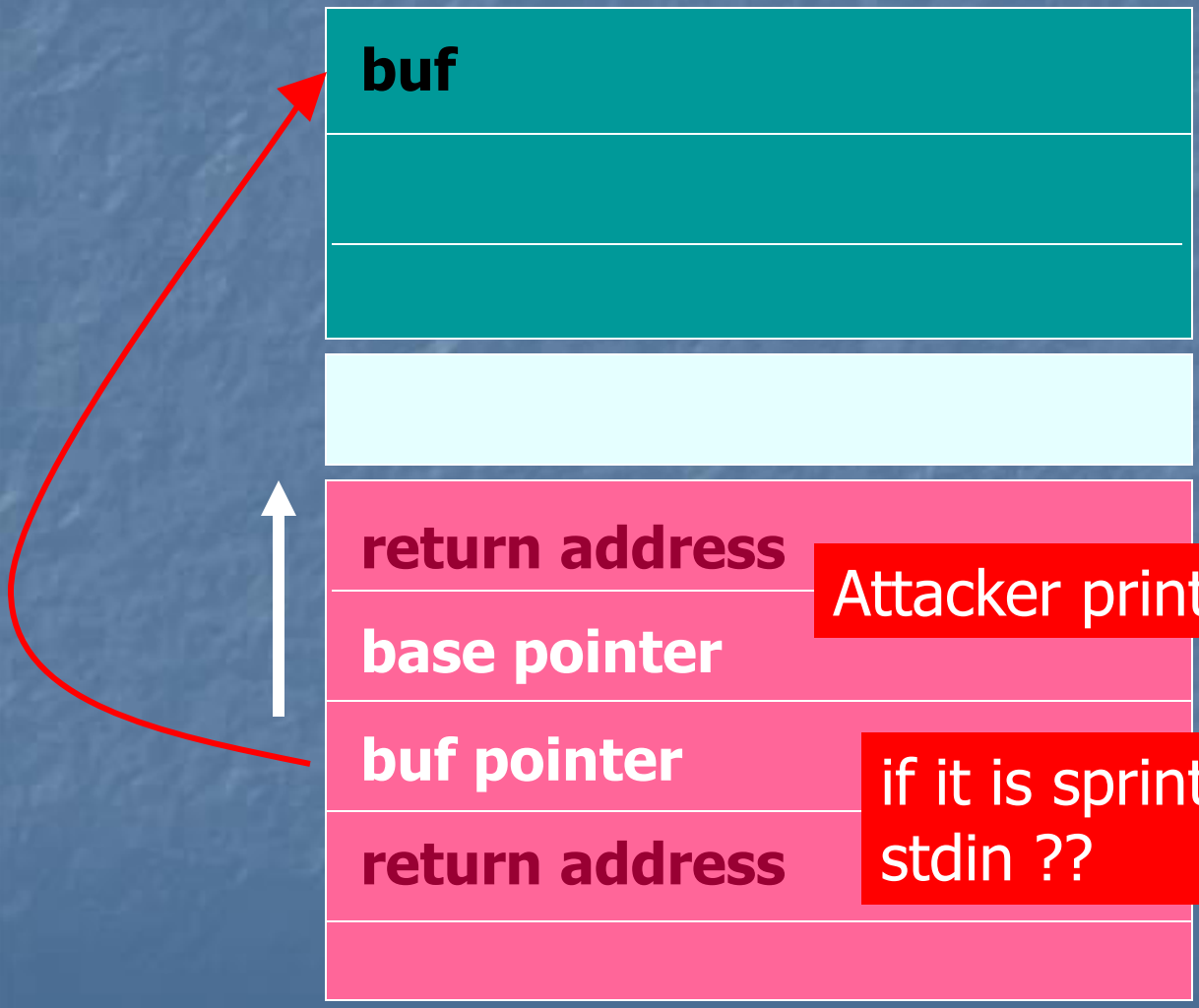
```
printf("%x%x%x%x%x%x%x%x%x");
```



Attacker prints addresses !!

What about sprintf ??

```
char buf[]= "%x%x%x%x%x%x%x%x%x";
printf(buf);
```



What do we control ?

Crash the program and dump the memory

- illegal address
- non-readable area

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

Viewing the process memory

- Viewing the stack
- Viewing memory at any location

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

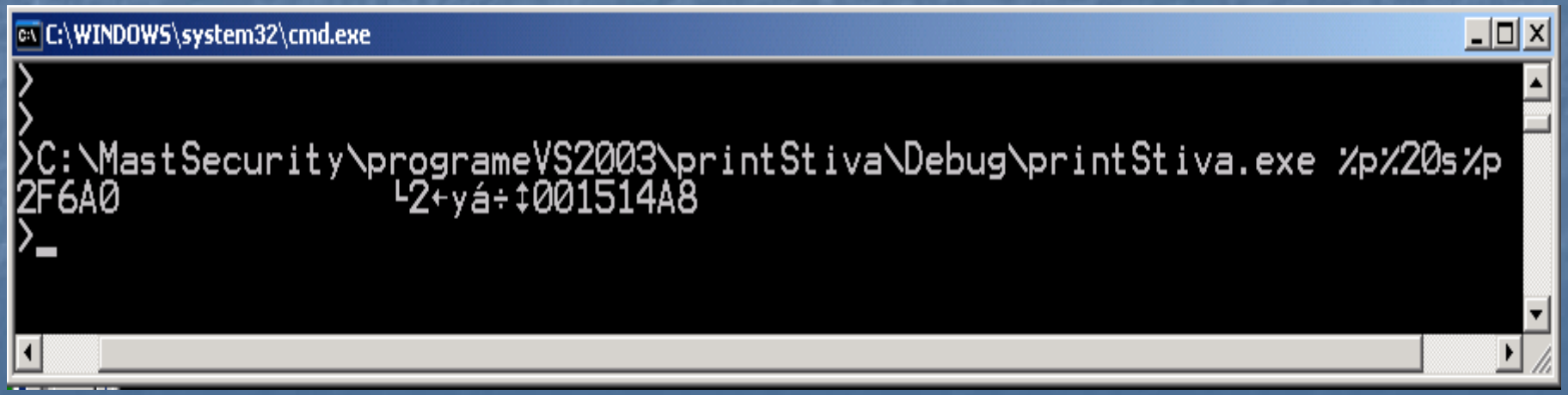
```
char buf[]= "%x%x%x%x%x%x%x%x%x%x";
printf(buf);
```

- Query parameters from the stack ? How manny ?
- Query data from anywhere in the program
- Crash the program by:
  - %s from non-readable memory.
  - %f with devide by zero

"Walk Ratio": The ratio between the pointer advance and the characters it take.
%x gives 1:4
%lf gives 1:8, but risks devide by zero.

A user supply format string gives the attacker a read only debugger access into the application!

```
#include <stdio.h>

void main(int argc, char *argv[])
        {
                int count = 1;
                while(argc > 1)
                        {
                                printf(argv[count]); printf(" ");
                                count ++;       argc --;
                        }
        }
```



scanf vulnerability

```
# include <stdio.h>
void main( )
{
        char buffer[250];
        scanf("%s", buffer );
        printf( "%s\n", buffer );
        getchar();
}
```

scanf("%249s", buffer);

%n – The dangerous option

- **%n** writes into the ***int*** pointed to by the respective argument the number of characters printed so far.

```
int count;
printf( "12345678901234567%n890", &count );
printf( "   Number displayed: %d\n\n", count );
```

%n – The dangerous option

- Attacker supply a pointer, **where and what to write.**

```
char buf[16]="Overflow this!%n";
printf(buf); …
```

- What if printf does **not have an argument**?

  Stack location pointed to by printf's internal stack pointer will be interpreted as address into which the number of characters will be written!

%n – The dangerous option

- Attacker supply a pointer (&count), **where to write.**
- By playing with field length, attacker can also control **what to write**.
- Bare shortcuts, that may require printing an average of 2GB of data.

```
int count;
printf( "12345678901234567%hn7890", &count );
printf( "   Number displayed: %d\n\n", count );
```

sprintf buffer-overrun error

```
int main( int argc, char *argv[] )
{
  char buffer[250];
  sprintf(buffer, "Usage: %s <name>\n", argv[0]);
  printf( buffer );
}
```

sprintf buffer-overrun error

- formatting user supplied arguments, or environment variables (registry).
- incorrect calculation of total buffer length when combining buffers.

```
bool SprintfEroare(int linie, unsigned long err, char* msg)
{
    char buf[132];
    if(msg == NULL) {          assert(false);          return false;     }
    sprintf(buf, "Eroare %d  in linia  %d - %s\n", err, linie, msg);
    return true;
}
```

gets - fgets

- gets data from stdin, rarely secure.
- Has no facility to check the buffer's length.
- Alternative

```
fgets( buffer, buff_size, stdin);
```

with identical results, but boundaries checking.

getchar

```
#include <stdio.h>
 void main()
{
        char buffer[250];
        int i,c;
        for( i=0; (c=getchar())!=EOF &&
                                        c!='\n' && i<250;
                                                        ++i )
                buffer[i]=c;
        buffer[i]='\0';
        printf("%s\n", buffer);
        getchar(); getchar();
}
```

getchar

- If the input length is 250 characters or more, a single byte after the end of the buffer is overwritten with **NULL**.
- With an upward growing stack, and a little endian machine (such as Intel), this means overwriting the LastSemnifByte of the pointer right after the buffer, with zero.
- With the buffer size occupying most (but not all) of the previous 256 block, there is a very high probability that the new pointer points back into the buffer.
- There is a good chance that this bug to be exploitable!

SQL Injection

- **Executing unauthorized SQL commands**

select * from client where nume = " + tbNume.Text

'Alina'

'Alina' or 1=1--

'Alina' drop table client -

'Alina' select * from users_pass --

SQL Injection - solutions

- **Quoting the input**
  - Valid only for *string variable*
  - Don’t solve the problem …

```
string frazaSQL =
     "select nume, telefon1, medie from doc2006 where nume =  '"
                                                  +txtNume.Text+"'";
string frazaSQL =
    "select cod, nume, telefon1, medie from doc2006 where cod = '"
                                                  +tbCod.Text+"'";
```

SQL Injection - solutions

- Using parameters

```
OleDbParameter param1;

string strSql =
  "select nume, telefon1, medie from doc2006 where nume = @numParam";
param1 = myCmd.Parameters.Add(
   new OleDbParameter("@numParam", OleDbType.LongVarChar) );
param1.Direction = ParameterDirection.Input;
myCmd.Parameters["@numParam"].Value = txtNume.Text;

OleDbDataReader dr = myCmd.ExecuteReader();
while (dr.Read())
{
        textBox1.Text+="\r\n"+dr["nume"]+ "    " + dr["telefon1"]+
   " + dr["medie"];
}
```

SQL Injection - solutions

Stored procedures

```
private void creareProc_Click(object sender, System.EventArgs e)
{
string strCreare = "CREATE PROCEDURE PREMIANTI AS "
+"SELECT nume, telefon1, medie FROM doc2006 WHERE medie>9.5 ";
OleDbConnection con = new OleDbConnection(
  @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\doctoranzi2006.mdb");
con.Open();  OleDbCommand myCmd    =new OleDbCommand();
myCmd.CommandType = CommandType.Text;     myCmd.Connection = con;
myCmd.CommandText="DROP PROCEDURE PREMIANTI";
OleDbDataReader dr;
try { dr = myCmd.ExecuteReader(); dr.Close();}
catch { MessageBox.Show("Procedura de sters nu exista!!");}
myCmd.CommandText = strCreare;
try { dr = myCmd.ExecuteReader();dr.Close();}
catch { MessageBox.Show("Procedura nu a fost creata!!");}
}
```

SQL Injection - solutions

Stored procedures

```
private void apelProc_Click(object sender, System.EventArgs e)
{
OleDbConnection con =
new OleDbConnection(
   @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\doctoranzi2006.mdb");
con.Open();

OleDbCommand myCmd          =new OleDbCommand("PREMIANTI",con);
myCmd.CommandType=CommandType.StoredProcedure;
OleDbDataReader dr = myCmd.ExecuteReader();
textBox1.Text="Lista premiantilor\r\n====================\r\n";
while (dr.Read())
 {
   textBox1.Text+="\r\n"+dr["nume"]+ "   " + dr["medie"]+ "  " + dr["telefon1"];
 }
dr.Close();
}
```

SQL Injection - solutions

- Using stored procedures

```
CREATE PROCEDURE MyProc @input varchar(128)
AS    exec(@input)
```

- drag /drop, without clearing clipboard ?

SQL Injection

- **Other vulnerabilities**
  - Standard user name / password: sa / as
  - Hardcoded passwords or conection string
- **Solutions ?**
  -

Protecting program code from reverse engineering

Reverse-engineering:

- **Disassemblers** turn an executable file into human-readable code - **assembler**
- **Decompilers** are designed to translate an executable file into a higher-level language, generally **C/C++**;

A. Obfuscation
B. Linking and Packaging
C. Application Watermarking

Decompilers

- www.remotesoft.com/salamander
- reflector.exe

A. Obfuscation

Definition

a collection of tactics that transform a program into a new program that functions the same as the original and wich is more difficult:

- for a **human to understand** its inner workings.
- for a **decompiler to output** correct, understandable, high level source code.

```
_(  __,  ___,
  _____){___/__<=1?_(__,___+1,___
  _):!(___%__)?_(__,___+1,0):___%
  __==___/
__&&!____?(printf("%d\t",___/__),_(
  __,___+1,0)):___%__>1&&___%_
  _<___/__?_(__,1+
___,____+!(___/__%(___%__))):__
  _<__*__?_(__,___+1,____):0;}mai
  n(){_(100,0,0);}
```

```
void primes(int cap)
{
   int i, j, composite;
   for(i = 2; i < cap; i++)
   {   composite = 0;
       for(j = 2; j < i; j++)
                   composite += !(i % j);
       if(!composite) printf("%d\t", i);
   }
}
int main() {   primes(50); }
```

- (1) function body is rewritten as a single while loop with a sequence of cascading if-else structures inside of it;
- (2) the while loop is then rewritten as recursion;
- (3) the cascading if- else statements is rewritten as a single conditional statement. The remaining transformations purposefully hide the details of the function by:
- (4) removing all intermediate variables;
- (5) renaming the few variables left to _, ___, etc.;
- (6) removing all whitespace and unnecessary parentheses

Obfuscation

An obfuscator should be able to transform debugging information along with the output programs.

- ex. PDB files for .NET that contain line number information

Incremental Obfuscation

- Import the original name mapping into the obfuscator and let it use the same names.
- Map file becomes an artifact that must be tracked throughout the component's lifecycle

Obfuscation techniques

I. Metadata Removal / Insert
II. Changing names and identifiers
III. String Encryption
IV. Control Flow Obfuscation

Obfuscation techniques

I. **Metadata Removal / Insert**

- **Removal:** Properties, Events, Method Parameter names, Some Custom Attributes / Annotations; comments; otherwise **provides hints** to decompilers and attackers.
- If it is a library, we can not remove metadata (who use it ?)
- **Inserting** invalid metadata into the application that the runtime does not use, but that the disassembler tries to parse.

Obfuscation techniques

II. Changing names and identifiers

- Classes, Interfaces, except classes referenced dynamically (that is dynamically loaded by name, based on a user input string).
- Fields, Methods; overloading / overriding ?
- Function args- same name for almost all !!
- Variables- all names used only internally can be renamed

Obfuscation techniques

Changing names and identifiers

- **If the program is a library:**
  - all external names can be renamed if **all clients are obfuscated in the same way**.
  - External names must be preserved if clients are not obfuscated. Most commercial obfuscators support both library scenarios with minimal configuration.
- **XML Serialization**: "serializes an object and uses the names of the members, as determined at runtime, as the XML element names."
  - rename and call them with the new name;
  - rename only if producers and consumers of the serialized objects use same renaming system

Obfuscation techniques

Changing names and identifiers

- 1:1 table
- 1 : n algorithms: use as few unique names as possible within any given scope ({ })
  - Requires deep analysis to implement, but more difficult to attack. Global search and replace are ineffective.
  - Size reduction (fewer names to store is a nice side effect.)

Obfuscation techniques

Changing names and identifiers

- **Standard Overload Induction** - Rename as many methods as possible to the same name (**50% of all methods in an application may be renamed to exact same name**, by overloading)

Obfuscation techniques

Changing names and identifiers

- Standard Overload Induction

```
float getCheckingBalance(int accountNumber)        { }
void setCheckingBalance(int accountNumber, float value)  { }
int getCheckingStatus(int accountNumber)  {  }
```

```
float a(int a)  { }
void a(int a, float b)  { }
int b(int a)  { }
```

Obfuscation techniques

Changing names and identifiers

- Enhanced Overload Induction

```
float getCheckingBalance(int accountNumber)        { }
void setCheckingBalance(int accountNumber, float value)  { }
int getCheckingStatus(int accountNumber)  {  }
```

return type become discriminatory !!

```
float a(int a){}
void a(int a, float b){}
int a(int a){}  // discriminatory return value
```

- Destroys original method overload relationships
- Decreases size of application
- Does not hurt performance

Obfuscation techniques

III. String Encryption

- **literals** and other **sensitive information**
- **fast** but **weak encryption** at runtime
- performance impact anyway !

Obfuscation techniques

IV. Control Flow Obfuscation:

The goal is to transform the original instruction sequence into a logically equivalent sequence that does not mirror the source.

Basic rules:

- loops are backward jumps
- if statements are forward jumps
- conditional operator (? :) is 2x faster than if – then - else

Obfuscation techniques

Control Flow Obfuscation

- procedures that never execute

```
int g = 0; a[0]=100;
for ( ; g<1000;++g)
 {
        if (a[g] > 50) {  break;  }
}
```

```
int g=0; a[0]=100;
for (; g<1000; ++g)
{
  if (a[g] > 50)
     {
          int y = 44;
          int x = y - (22*2); // Splitting blocks of instructions and
          if (x==0) break;   // introducing spurious control flow
      }
}
```

Obfuscation techniques

Control Flow Obfuscation

- loop count from 331 to 340 instead of 1 to 10
- procedures having loops count backwards **(--i)**

```
int i = 0;
for ( i=331; i<341;++i)
 {
        a[i-330] = i-330;
}

i=1; while (--i) { x=Y*12-v[j-k] ; }  // ??
```

Obfuscation techniques

Control Flow Obfuscation

- computed loop count from 11 to 8003 instead of 1 to 1000

```
int i = 11;
while(i<8003)
 {
        a[(i-3)/8] =(i-3)/8 ;
        i+=8;
}
```

Obfuscation techniques

Control Flow Obfuscation

- replacing a **reference** to a variable with a programming construct designed to use a second variable **to replicate the reference** to the first variable
- generating a second **thread**;

Obfuscation techniques

Control Flow Obfuscation: *smoke* & *mirror*

- inserting code that never execute;
- inserting one or more unnecessary statements / code is executed but is useless)

```
int i = 0, x=10;
if(x>0) goto et100;
while(i<0)
{
  et100:   printf("Do something");
           i++;
 }
```

Obfuscation techniques

Control Flow Obfuscation : *spaghetti code*

- **flow control abstractization**:
  transform a block in an equivalent sequence

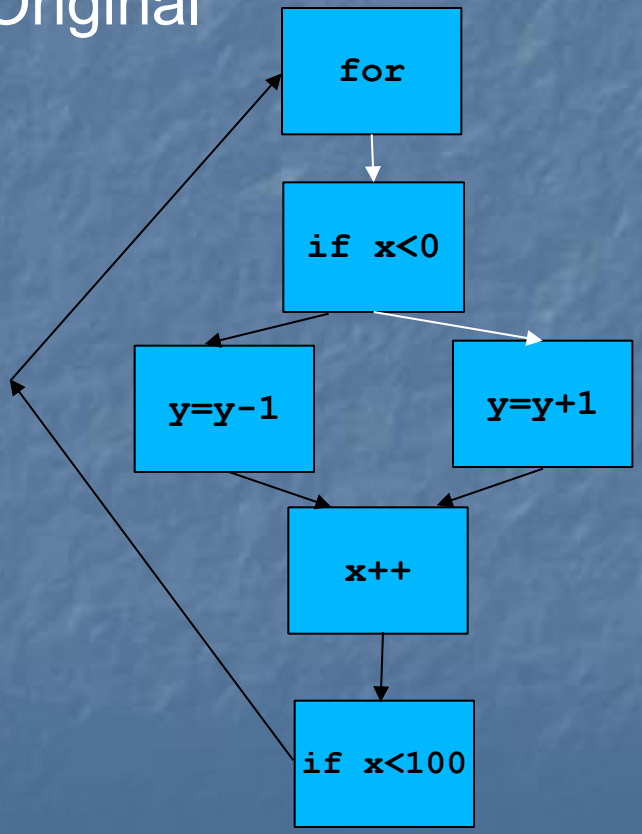
ex. *for* translated into *if* + *go to*;

```
int i = 0;
for(; i<10; i++) printf("%d",i);


int i; i = 0;
etStart:
        if(i>=10) goto etFinal;
        printf("%d",i++);
        goto etStart;
etFinal:
```

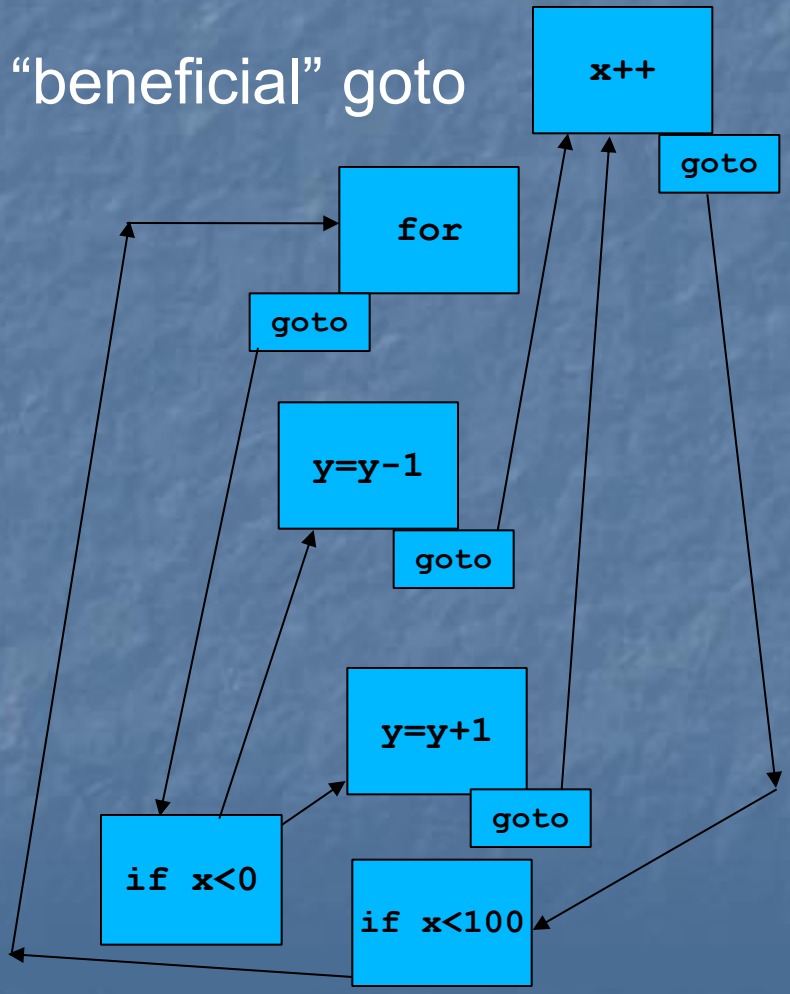
Obfuscation techniques

Control Flow Obfuscation: *block re-ordering*

Original



"beneficial" goto



Obfuscation techniques

Control Flow Obfuscation:

high level language breaking

including assembler statements to break asociated context from high level language

```
int i = 0;
for(i=0;i<10;i++) printf("%d",i);
_asm
        {
                add ax, bx;
        }

f(x,y);
```

- sequence with no equivalence in the high level language

Obfuscation techniques

Using opaque constructs

- **opaque construct:**
  - depends, at least in part, on a value of a predefined variable;
  - code sequence generate a predictible flow for obfuscator

```
int v; a=5; b=6;
if(x<0)
        { /* code where a and b remain unchanged */}
if(b<7) a++;

if(Math.Random() <0) { /* secventa inutila */}
```

Obfuscation techniques

Obfuscation by code aggregation

- **inlining - outlining:**
  - aggregated method code is dispersed in the class as inline functions; a new method call dispersed code in one unit
  - virtual functions can be inlined ? Yes, but co-exist both versions (**inline** – for object based call, **outline** – for pointer / reference based virtual call)
- **enforced aggregations:** unrelated code parts are aggregated in the same method
- interleaving programming statements from different subroutines.

Obfuscation techniques

Obfuscation by data aggregation

alters how data is grouped together (ex. A two-dimensional array can be converted into a one-dimensional array and vice-versa; **changes the way in which program statements are grouped** together

- restructuring an array so that it has a different number of dimensions
- merging two variables into a single variable /reusing/;
- splitting a variable into at least two variables;
- replacing data with a call to a procedure that produces the data;
- generating data structure grouping elementary data; union in C++;

Obfuscation techniques

Obfuscation by data aggregation

- define a second class and a third class, the third class inheriting directly from the second class, and the second class and the third class being designed to replace the first class;
- removing the first class from the computer program

Optimization as Obfuscation

Source code:

```
int temp = a;
a = b;
b = temp;
```

Compiles as

```
push a
pop temp
push b
pop a
push temp
pop b
```

Which can be optimized to:

```
push a
push b
pop a
pop b
```

And commonly wrong decompiles to:

```
a = b;
b = a;
```

```
int a=1, b=2; a^=b^=a^=b;  // !!!
```

Optimization as Obfuscation
Loop unrolling / splitting

Source code:

```
for(int i=0; i<1000; i++)
        a[i] = b[i] + c[i];
```

can be optimized to:

```
for(int i=0; i<1000; i+=2)
{
   a[i] = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
}
```

Double unrolling ?

```
for (i = 1; i <= 6000; i++)        a[i] = a[i] * b + c;
```

Generalization ?

Pentium® 4 / Intel® Xeon™ can correctly predict the exit branch for an inner loop with 16 or fewer iterations, if that number of iterations is predictable

Optimization as Obfuscation

- **Caching local variables on the stack**; using the cache when a function has many recursive calls capable to be stored in **level1**; difficult to know the number of calls and if all calls can be stored; otherwise are stored the last ones and restored the others ...

Obfuscation - conclusions

- Make **debbuging harder** and forces us to keep in parallel the original
- High **dependence on platform** and language features
- **conflict with reflection;** inquery at runtime
- can rename a class and then use it under the new name, but with the price of high complexity for obfuscator

Obfuscation - conclusions

- **additional consumption of resources**
- reduce the uncontrolled distribution of the code (nothing to do with a encrypted code), but also minimizes the chance of development by others (do just for himself !)
- maximum level of obfuscation without losing control ? (how far deserves to complicate the code?
- price paid: keep copies of all versions in the program development...

B. Linking and Packaging

- **Dotfuscator:** obfuscation starts from the compiled code (IL), not from source code so it does not affect the source debugging

- **Assembly Linking** – taking over existing assemblies and mixing them to produce a **single assembly (EXE +DLL) !**
- **Assembly Linking + Pruning --> extract from DLL only necessary methods !**

C. Application Watermarking

- **Embedding and hiding data inside an application;**
- **Ex.: serial number** helps to see who has allowed leaks of information and ease find the attackers

Vulnerabilities in OOP: bad casting

Casting from srs to dst

```
#include <iostream.h>

class dst { public: int x; };

class srs
{
    private: int x;
    public:
        srs(int a):x(a){}
        operator dst() { dst d; d.x=x; return d; }
};

void main()
{
    srs s(1); dst d=(dst)s; cout << d.x;
}
```

gives access to private data.

Using *private* constructors

```
public class Capability
   {
      private int capabilities;
      static private int specialCapability=4;
      private Capability(int c) { capabilities = c; }
      public bool Test(int capnum)
                  { return (capabilities & (1 <<
     specialCapability) )!= 0; }

      static public Capability Create(int c)
           { return new
                Capability(c & ~(1 << specialCapability)); }
   }
```

Using *private* members with restrictive accesors (*set* allowing just one setting)

```
class Uid
{
 private:  int id;
 public:
          Uid():   id (0) { }
          int get() { return id; }
          void set(int new_id)
                    {
                     if (id == 0)
                              id = new_id;
                     else
                              throw new SetException();
                    }
};
```

Vulnerabilities in OOP: Virtual Methods vulnerabilities

- vtable; vptr
- Countermeasures:
  - placing them before member variables in memory
  - what if several objects are allocated in a contiguous memory ?

Vulnerabilities in OOP: unsafe plugins

(Abadi and Fournet, 2003)

```
namespace ConsoleApplication1
{
   abstract class Trusted
   { // full privileges
      static protected String tempfile = "/tmp/tempfile";
      abstract public void proceed();
     static  void Main()
      {
         BadPlugin bp = new BadPlugin();
        try { bp.proceed();     }
        catch (Exception e){ File.Delete(Trusted.tempfile); throw(e);}
      }
   }
   class BadPlugin : Trusted
    { // low privileges
      override public void proceed() { tempfile = "/etc/passwd"; }

    }
}
```

Who cares about security?

Businesses do:

Feb 2000:

- Yahoo, Buy.com, Amazon.com, CNN, etc. shutdown by massive DDoS attack. Yahoo lost more than $1m per minute...

Aug 2000:

- Fake news report posted on internet news agency computer - "Emulex Corporation"'s CEO resigns and quarterly earnings adjusted to loss, not profit. Share price drops 60% in hours ($billions!).

sandbox model - security technique is to run applications from unsafe sources through an isolated environment (sandbox) so that it can be tested without any breach of privilege affecting the original system. This isolation can be:

- **a virtual machine** providing validation for data types and authorization for access to memory;
- **a native API** controlling access to resources. In this case, methods like SecurityManager.**checkPermission** (checkRead, checkAccept, ...) or similar variants are called before any sensitive operation.

What is the .NET Framework?

- Microsoft's cross-language development platform
  - Execution environment / VM:
    Common Language Runtime (CLR)
  - Intermediate Language: MSIL (similar to bytecode)
  - Class libraries: "The Framework" (FCL/BCL)
  - Language compilers (+30, MS & 3rd party)
  - Development tools: Visual Studio.NET

- Any language that compiles to MSIL can use and extend the .NET Framework

- CLR provides a secure execution environment for "partially-trusted code"

Vulnerabilities in Virtual-Machine Environments

- **bytecode** in Java and **IL** in .NET; run on multiple processors ==> **compiled** code would be simpler, and therefore easier to understand.
- **unmanaged** C++ will not be secure when written as a desktop application.
- **ILDASM.EXE** - which provides well-formatted IL code fulfilled with metadata
- **Countermeasures:**
  - either unmanaged (non-IL) code,
  - or to have **all valuable code run on servers** and expose only interfaces, so there is no chance of anyone seeing the IL code /see validations on web/

Vulnerabilities in Virtual-Machine Environments

- DLL hell
  - Incompatible versions
  - DLL stomping
  - Incorrect COM registration
  - Shared in-memory modules
  - Lack of serviceability
- strong names that consist of the file's simple text name, version number, and culture information, plus a public key and a digital signature

Vulnerabilities in Virtual-Machine Environments

- the application contains hints about IT infrastructure (e.g. SQL statements, DB connections strings, etc.
- Don’t store secrets in config file !
- Avoid writing software that requires admin privileges !

Vulnerabilities in Scripting Languages

- protecting source code is a vital part of information security (without access to source code).
- **scripting languages are not compiled => the source code is revealed**
  - client side scripting for validation
  - reduce round-trips for sensitive data
- having validation routines in clear text can tell an attacker the kind of input that a server cannot handle

Vulnerabilities in Scripting Languages

- .NET config files are stored as XML !
- "hidden fields" to echo pricing data to a web browser, usually in the absence of another method of maintaining state
- Promote **web services and server-based programming**

Vulnerabilities in Scripting Languages

- **XSS "Cross Site Scripting"** - vulnerability that allows the injection of executable script code in a variable or an input unfiltered field, on a web page from a server

Vulnerabilities in Scripting Languages - XSS

```
<IMG SRC="javascript:alert('XSS')"
<SCRIPT>alert("XSS");//<</SCRIPT>
<IMG """><SCRIPT>alert("XSS")</SCRIPT>">
<DIV STYLE="width: expression(alert('XSS'));">
<IMG STYLE="xss:expr/*XSS*/ession(alert('XSS'))">
<SCRIPT>window.location.href= "http://www.ase.ro
    "//<</SCRIPT>
```

validateRequest=false

Test:

http://localhost/stareSesiune2/default.aspx

Key Components of the .NET Security System

- Type safety and verification
- Permissions, demands and stack inspection
- Policy/trust management system for assigning permissions to assemblies
- Application deployment model

Type Safety & Verification

- CLR provides memory protection through type safety verification
- Every MSIL method is type-safe verified before being allowed to be called by another method
- Verification happens as part of JIT compilation to binary
- The right to run unverifiable code is governed by a security permission (like any other privileged operation)

Strong typing

Strong typing: strict enforcement of type rules with no exceptions. All types are known **at compile time**, i.e. are **statically bound**. With variables that can store values of more than one type, incorrect type usage **can be detected** at run time

- An **integer** is not a **pointer** (reference).
- A **byte array** is not a **function**.
- An array / string cannot be accessed **beyond its bounds**.
- All variables are **initialized before** being read

Strong typing

Can be enforced

- **dynamically** (ill-typed program halts)

check types during execution; **tag data with their types**.

Example: **when evaluating** x + y, **check** that x and y contain integers; compute sum and **tag** it as integer.

- **statically** (ill-typed program is rejected at compile-time).

check types at compile-time, using static analysis; execution proceeds without type tests, on untagged data.

Example: **when compiling** x + y, check that x and y have been declared as integers; record that this expression has type int.

Strong typing does not cover bad casting

Casting from:

class C { int x; }

to

class D { byte[ ] a; }

causes **pointer *a*** to be forged from **integer x.**

.NET security

- **security mechanism** with two general features:
  - **Code Access Security**
  - **Validation and verification**
- **Code Access Security uses evidence to determine the permissions granted to the code**
- Code that performs some privileged action will make a **demand** for one or more permissions

.NET security

- **validation**: CLR checks that the assembly contains valid metadata and CIL, and whether the internal tables are correct
- **verification**: checks to see if the code does anything 'unsafe'
-

- **Assemblies** are the core unit of code development and distribution
  - Similar to a .DLL or shared library
- **Assemblies** are the **minimum code unit** that have identity

- **Permissions** are granted on an **assembly - wide** basis
  - All methods in the same assembly have the same rights

- **Applications** are **collections of assemblies** dynamically assembled based on referenced classes

Evidence

- **Application directory**
- **Publisher** - assembly's publisher's digital signature
- **URL**- the complete URL where the library was downloaded from.
- **Site** - hostname of the URL.
- **Zone** - defined security zones (defined by browser)
- **Hash**- a cryptographic hash of the assembly, which identifies a specific version.
- **Strong Name** - the X.509 certificate uniquely identifying a publisher

Policy

set of expressions using evidence to determine code group membership, which gives a permission set for the assemblies

- **Enterprise** - policy for a family of machines that are part of an Active Directory installation.
- **Machine** - policy for the current machine.
- **User** - policy for the logged on user.
- **AppDomain** - policy for the executing application domain.

Permissions, Demands & Stack Inspection

- A permission is a set (or subset) of capabilities with respect to a resource. Ex: FileIOPermission(READ, "c:\")
- Most permissions are code-access permissions and implement stack-walking semantics
- A demand for a code-access permission must be satisfied by grants to every stack frame above the demanding frame
- Stack-walking is a defense against luring attacks

  Less-trusted code tricking more-trusted code into performing protected operations

Stack Inspection

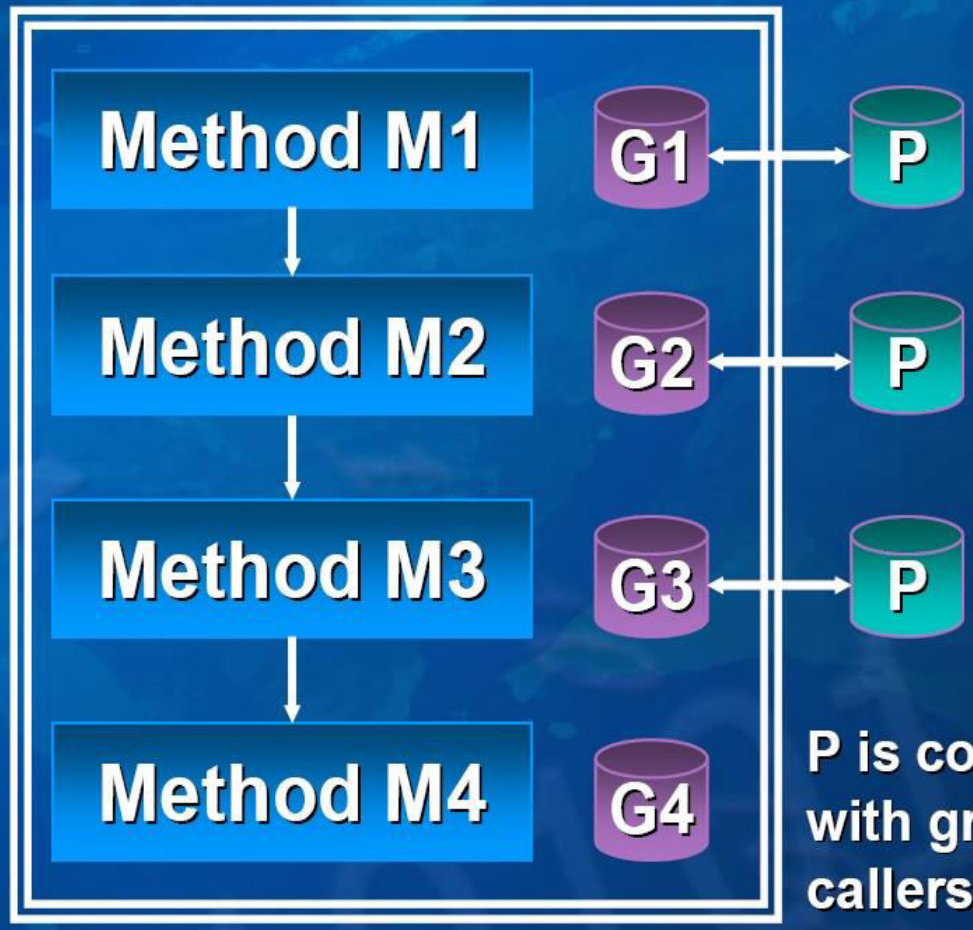
- Is grouping techniques which controls how methods call each other, ensuring that a method with reduced privileges can not indirectly benefit of extensive privileges of another method, by calling it
- **File permissions**: ***read, write, execute, delete*** **(pentru fişiere).**
  - **Socket permissions**: ***connect, accept connections*** **(pentru server gazdă).**
  - **Runtime permissions**: ***exit VM, load native code, define own class loader, define own security manager,*** **pentru lansarea în execuţie a unui task.**
  - **GUI permissions**: ***access clipboard, read pixels on screen*****, pentru interfeţele cu utilizatorul.**

- Assume that a function, even in the library, operate differently when it was called from secure code (such as browser) or an unsafe code (such as an applet).

Stack-walking Semantics

Each method has a set of corresponding grants

Call Stack Grows Down



Method M4 demands a permission P

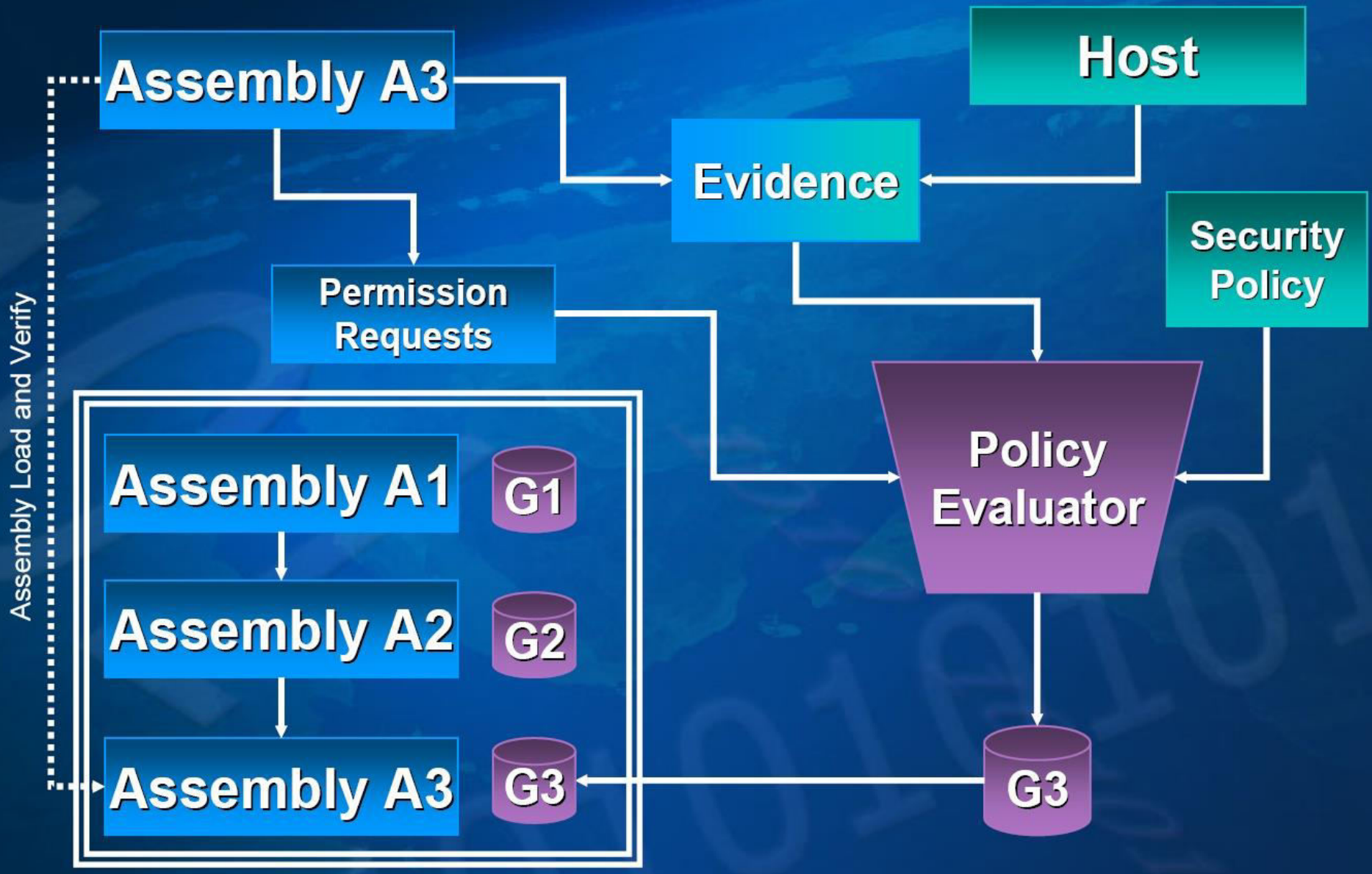
P is compared with grants of all callers on the stack above M4

Policy Evaluation in the CLR

- **Security policy evaluation** is the process of determining the **set of permissions** to grant to code based on **evidence** known about that code
- **Evidence** is typically info about a code assembly (e.g. publisher identity, URL)
- We assign permissions to assemblies in the CLR, just as we assign rights to groups of users.

```
Assembly thisAssembly =
                        Assembly.GetExecutingAssembly();
Evidence ev = thisAssembly.Evidence;
textBox1.Text="Host Evidence: ";
IEnumerator enumerator = ev.GetHostEnumerator();
while (enumerator.MoveNext())
{
    textBox1.Text+=enumerator.Current + "\r\n";
}
textBox1.Text+="\r\n";

textBox1.Text+="Assembly Evidence:";
enumerator = ev.GetAssemblyEnumerator();
while (enumerator.MoveNext())
{
    textBox1.Text+=enumerator.Current + "\r\n";
}
```

.NET Security in One Slide



Memory management in a GC'ed environment

- Usually, developers don't have to explicitly manage memory in a GC'd environment, but that's not quite true: they have to worry about manually clearing sensitive data as soon as they are done with it

- SecureString: specialized class for handling secrets like passwords (V2 feature)

- Proper use has impact throughout the class libraries (e.g. GUI, XML processing, etc.)

Memory management in a GC'ed environment

Explicit deallocation:

- Free the block *A*, keep the reference *A* around, wait until memory manager moves *B* into *A*'s spot.

Overriding

- Writing well-chosen integers or references in ***vtable*** allows arbitrary code to be executed.

Get the App Model Right

- A single API set/framework/SDK can support multiple application models
- The corresponding security system has to work for all of these models
- But it should best support and facilitate the model that developers use most
- CLR V1 security was aimed at rich clients built from a soup of assemblies from multiple sources
- The developers don't do this – they write single-source applications

The "ClickOnce" Model

- An "application-based model" built on top of V1 security primitives
- Key features added for V2
- An "application" now has an identity of its own collection of assemblies + metadata, defined by an XML manifest
- Applications are self-describing
- The application becomes the common unit of code deployment, not the assembly
- Applications that run in the sandbox with the default set of permissions "just run"
- Applications that require "elevated" permissions (more than the default) have to declare additional necessary grants in the manifest
- User consent required to run, but decisions can be persisted based on code signer identity

Some common reasons why the users turn off or ignore security system

- Too hard to develop in the security model.
- The system is too hard to administer;
- Can't easily figure out the correct security policy or map it to real-world requirements
- The system is too restrictive – it doesn't let them get their work done

Axiom 1 (Murphy) All programs are buggy.

Theorem 1 (Law of Large Programs) Large programs are even buggier than their size would indicate.

Corollary 1.1 A security-relevant program has security bugs.

Theorem 2 If you do not run a program, it does not matter whether or not it is buggy.

Corollary 2.1 If you do not run a program, it does not matter if it has security holes.

Theorem 3 Exposed machines should run as few programs as possible; the programs that are run should be as small as possible.