Secure Programming

Jonathan Dechaux jonathan.dechaux@esiea.fr

Ecole Supérieure en Informatique, Electronique et Automatique
Laboratoire de Cryptologie et de Virologie Opérationnelles
38 rue des docteurs Calmette & Guerin, 53000 Laval France

Your exam will be in two parts:

- The two practices who will be done this three days : reports.
- The final exercice : report and a fully commented source code.

You need to send the corrected exercice and the complete report with all explanations and arguments about your choices.
You need to do an archive, formatted like this:
LastName_FirstName_SEC_secprog_2015.zip.
It need to be send to the following email address:
jonathan.dechaux@esiea.fr.
End of submission: 30/10/2015 23:59.

What is the best way of secure programming?

- You and your knowledge are the best in this case.

Tools

There are multiple ways and tools to do secure programming :

- Static analysis.
- Code checking.
- Dynamic analysis.
- Reverse Engineering.
- Flow control.

Is that enough to do secure programming?

- No, be aware on one thing, mistakes are inevitable.

Finding bugs

- Dynamic testing
- Multiple tests in different conditions
- Dealing with the QA team

Off-by-One Length Miscalculation

```
int get_user(char *user)
{
   char buf[1024];
   if(strlen(user) > sizeof(buf))
      printf("error: user string too long\n");

   strcpy(buf, user);

   ...
}
```

Off-by-One Length Miscalculation

- Strlen function involved.
- Doesn't count the NUL terminating character.

Copy with No Control

- Strcpy function involved.
- Doesn't verify the size of the two buffers.

Function pointer example

```
void good_function(const char *str) {...}
int main(int argc, char *argv[])
{
   static char buff[BUFFSIZE];
   static void (*funcPtr) (const char *str);
   funcPtr = &good_function;
   strncpy(buff, argv[1], strlen(argv[1]));
   (void)(*funcPtr)(argv[2]);
   return 0;
}
```

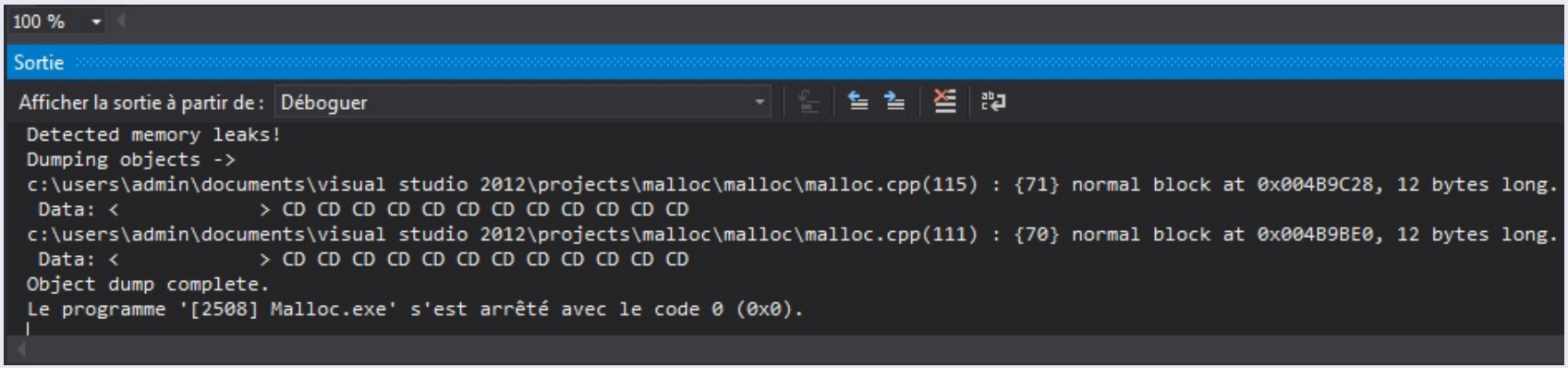
Program vulnerable to buffer overflow in the BSS segment

The BSS segment is the data segment where static variables are stored. *buff* and *funcPtr* are both uninitialized.
The call to *strncpy()* on line 7 is an example of an unsafe use of bounded string copy function. A buffer overflow occurs when the length of *argv[1]* exceeds *BUFFSIZE*.

Memory leak example

```
inBuf = (char*)malloc(bufSz);
if(inBuf == NULL)
   return -1;
outBuf = (char*)malloc(bufSz);
if(outBuf == NULL)
   return -1;
```

Visual Studio memory leak checking



Signed Integer Vulnerability Example

```
char *read_data(int length)
{
   char *buf;

   if(!(buf = (char*)malloc(MAXCHARS)))
      return -1;

   if(length < 0 || length + 1 >= MAXCHARS)
      return -1;

   if(read(sockfd, buf, length) <= 0)
      free(buf);

   buf[length] = '\0';

   return buf;
}
```

Signed Integer Vulnerability Example

A value of 0x7FFFFFFF passes the first check (because it's greater than 0) and the second length check because 0x80000000 is a negative value (it's lower than MAXCHARS).
The function read() would then be called with an effectively unbounded length argument, leading to a potential buffer overflow situation.

Stretchable buffer

```
char outbuf[512];
char buffer[512];
sprintf(buffer, "ERR Wrong command: %.400s", user);
sprintf(outbuf, buffer);
```

Stretchable buffer overflow

The *sprintf()* call on line 3 cannot be directly exploited because the %.400s conversion. This same call can be used to indereclty attack the *sprintf()* call on line 4 for example with the following value:

%497d\x3c\xd3\xff\xbf<nops><shellcode>

First thing

Don't over react and do multiple and multiple tests inside the source code.

Work hard in the team world

Trust you and trust your developpers team. Be aware of each part of the source code, and know what is the purpose for each part.

Three levels of protection

- Hardware protection : NX bit.
- Kernel protection : ASLR.
- Compile-time protection : GS cookie.

Severals tools

- Static analysis (Coverity, Code Sonar, ...).
- Dynamic analysis (Visual Studio, Valgrind, ...).
- MSDN and "trusted" sources.

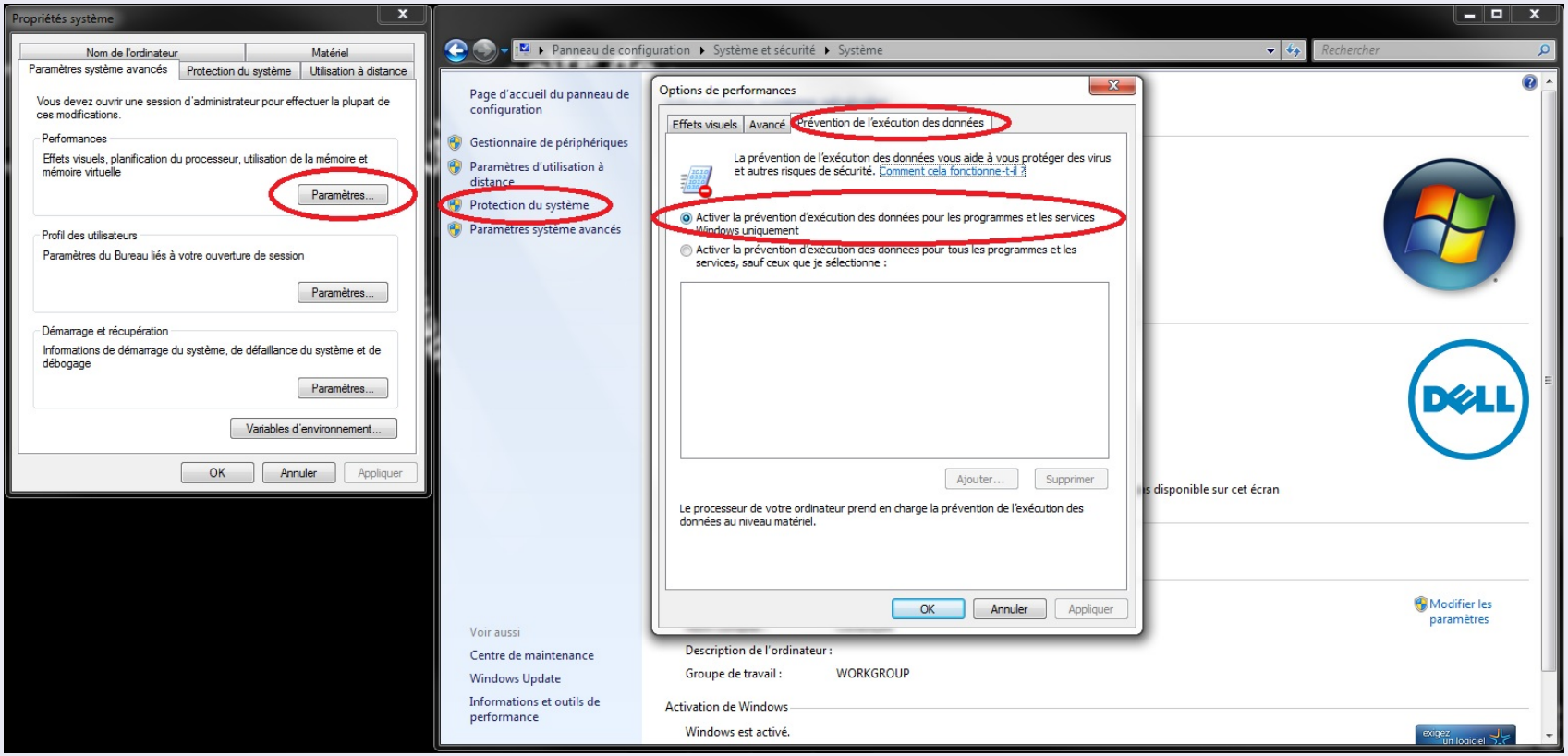
NX bit

- NX Bit stands for No Execute.
- It refers to bit number 63 (i.e. the most significant bit) of a 64-bit entry in the page table.
- An operating system with support for the NX bit may mark certain areas of memory as non-executable. The processor will refuse to execute any code residing in these areas of memory.
- executables have to be specifically linked to be NX-compatible (/NXCOMPAT).

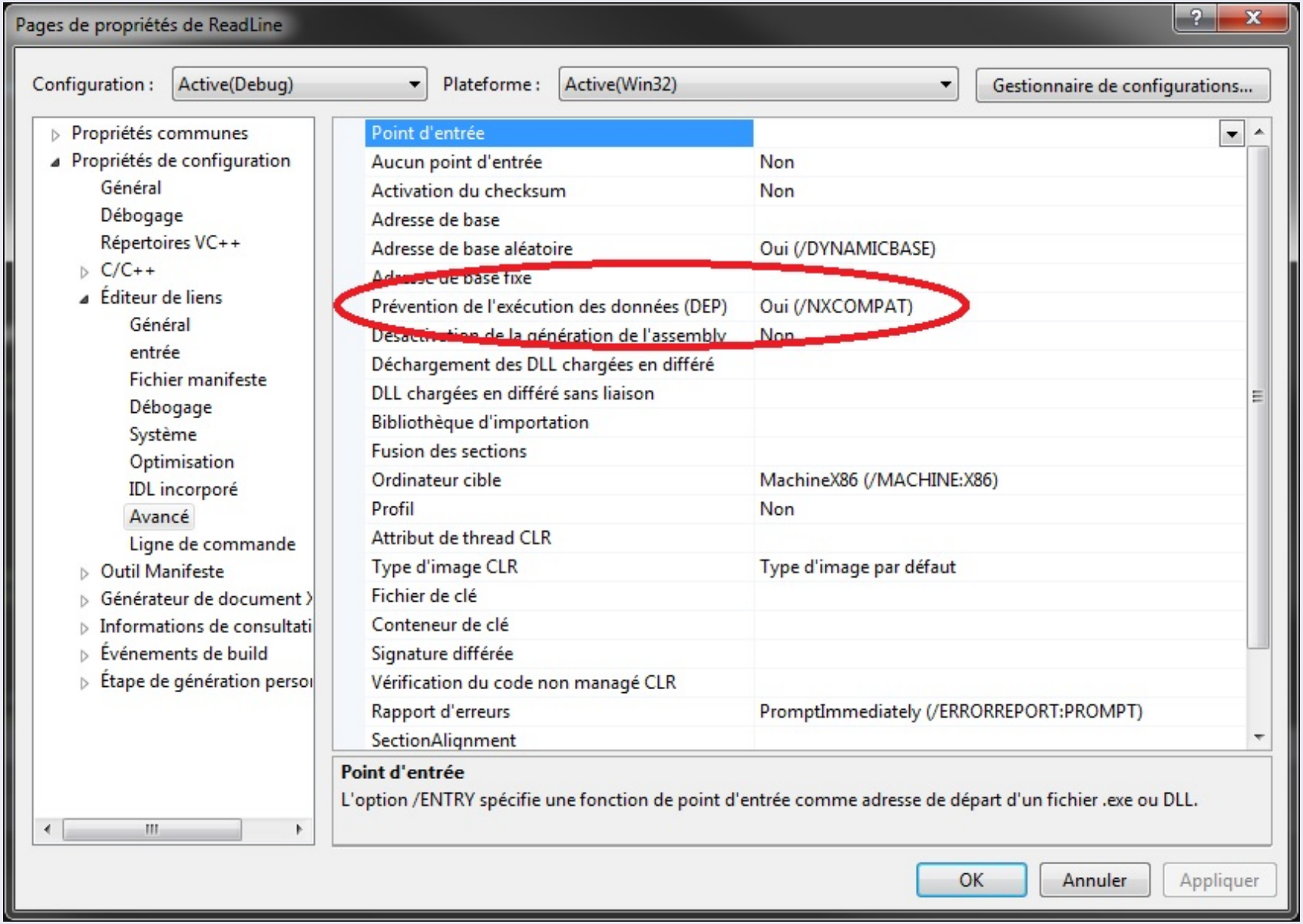
Activation / Deactivation in Command Line

- bcdedit.exe /set current nx AlwaysOn
- bcdedit.exe /set current nx Optout (Activated for the OS and all processes)
- bcdedit.exe /set current nx Optin (Activated for the OS components)
- bcdedit.exe /set current nx AlwaysOff

Activation / Deactivation in Windows Interface



Activation / Deactivation in Visual Studio



Integration in material development and other names

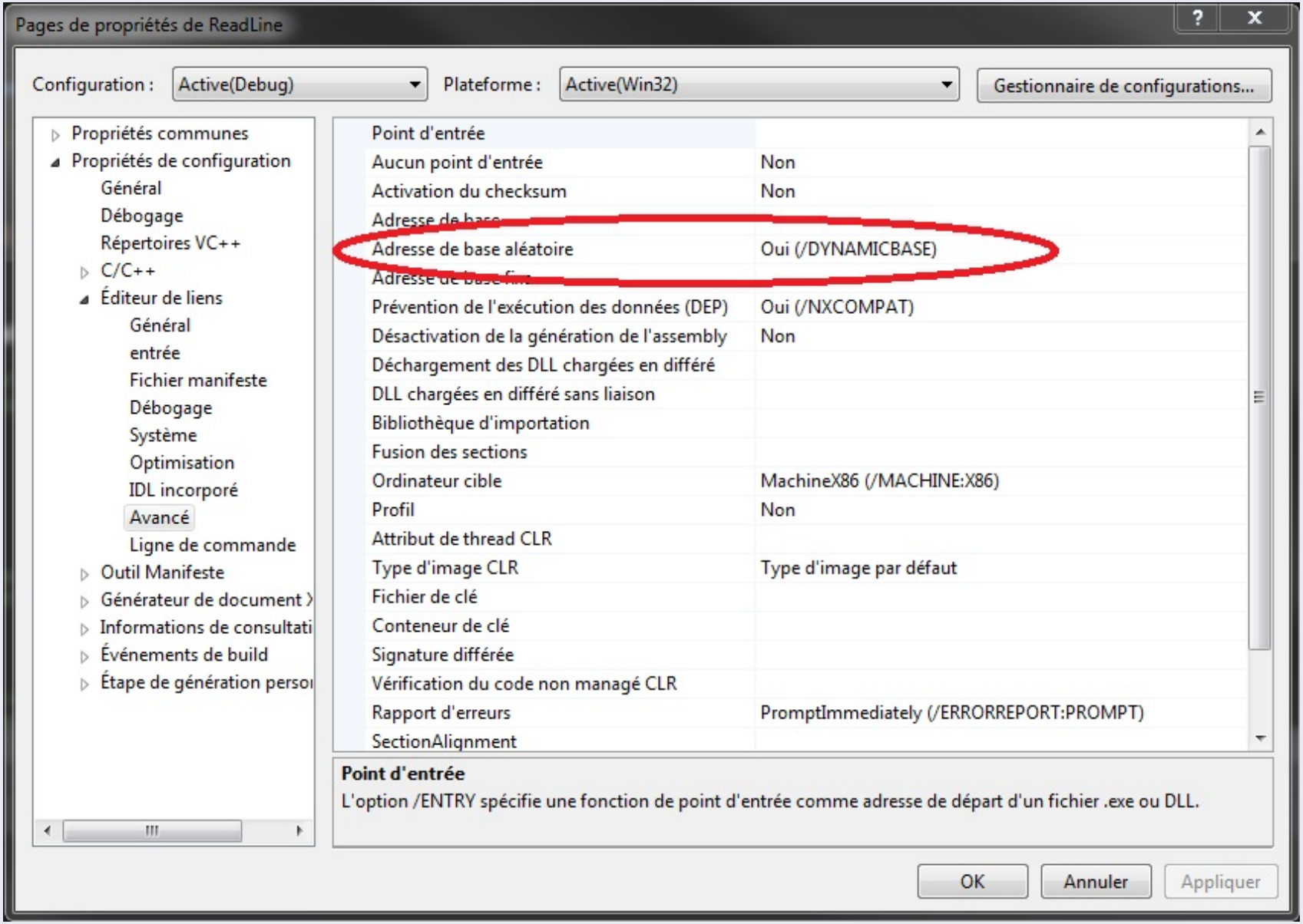
- Intel markets the feature as the XD bit, for eXecute Disable.
- AMD uses the names NX and Enhanced Virus Protection.
- The ARM architecture refers to the feature as XN for eXecute Never.
- This technology is named DEP by Microsoft, Data Execution Prevention.

ASLR

- Address Space Layout Randomization.
- Computer security technique which involves randomly arranging the positions of key data areas in a process's address space:
  - Base of the executable.
  - Position of librairies.
  - Heap.
  - Stack.
- Enabled by default since Windows Vista and Windows Server 2008, although only for those executables and dynamic link libraries specifically linked to be ASLR-enabled.
- ASLR for all executables and libraries is found in the registry at:

```
HKLM\SYSTEM\CurrentControlSet\Control
\Session Manager\Memory Management
\MoveImages
```

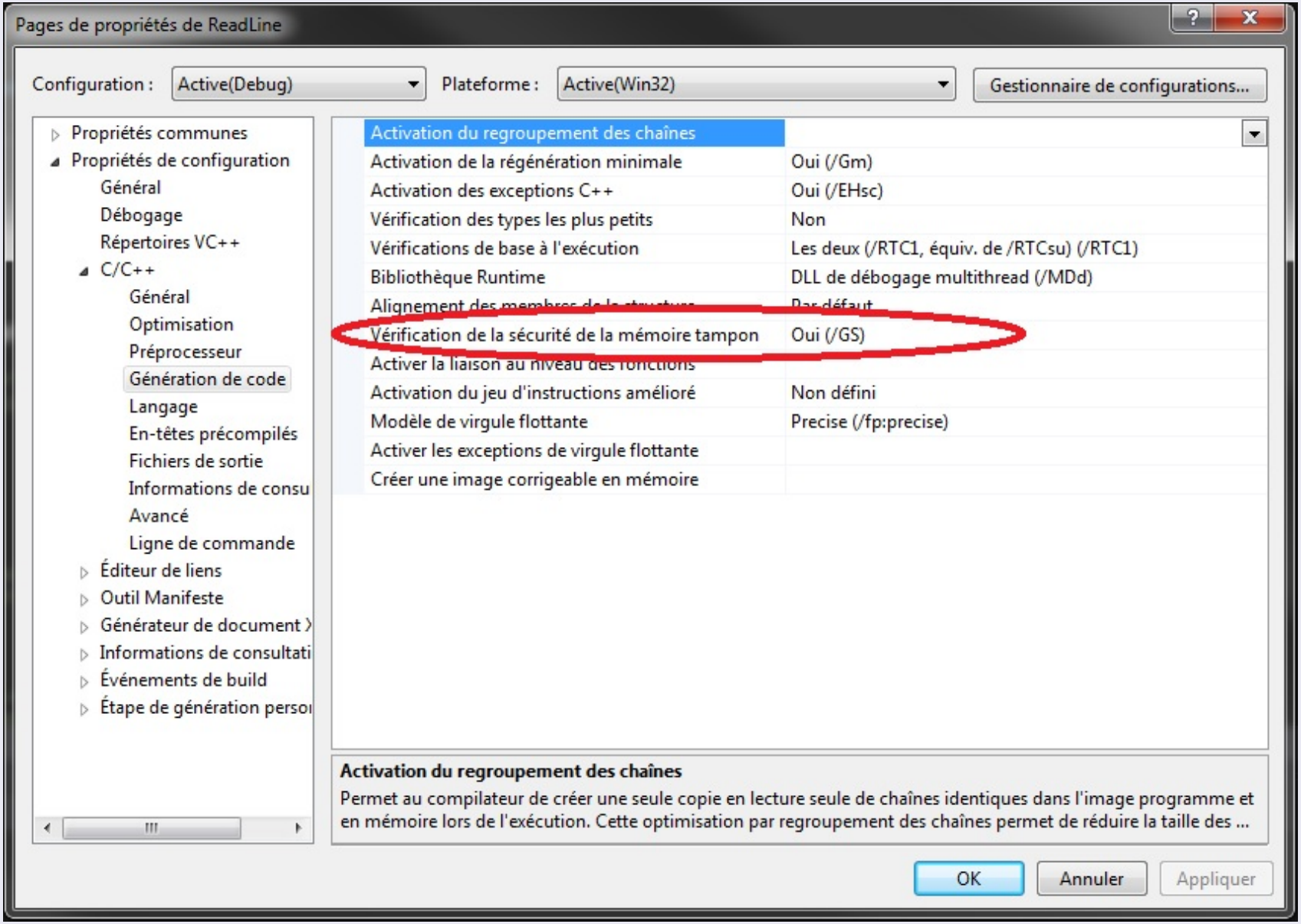
ASLR in Visual Studio



GS cookie definition

- MSDN definition : Detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters. Causing a buffer overrun is a technique used by hackers to exploit code that does not enforce buffer size restrictions.
- http://msdn.microsoft.com/en-us/library/8dbf701c.aspx

GS cookie in Visual Studio



Static Analysis

- Don't check your executable, only the source code.
- Don't check for memory leaks.
- Programmation language dependant.
- Coverity example.

Example 1

```
int main(void)
{
   File * f;
   int lSize;
   char *buffer;

   f = fopen("test.txt", "rb");
   fseek(f, 0, SEEK_END);
   lSize = ftell(f);
   rewind(f);

   buffer = (char*)malloc(lSize+1);
   for(i=0; i<lSize; ++i)
      fread(&buffer[i], 1, 1, f);

   printf("%s", buffer);
   return 0;
}
```

First Part : Uninitialized variables

```
File * f;
int lSize;
char *buffer;
```

Correction

```
int main(void)
{
   File * f = NULL;
   int lSize = 0;
   char *buffer = NULL;

   ...
}
```

Second Part : Tests and dealing with errors

```
f = fopen("test.txt", "rb");
fseek(f, 0, SEEK_END);
```

Correction

```
int main(void)
{
   f = fopen("test.txt", "rb");
   if(f != NULL)
   {
      if(fseek(f, 0, SEEK_END) == 0)
         ...
      else
         return 2;
   }
   else
      return 1;
}
```

Third Part : Variables type

```
int lSize;
```

Correction

```
int main(void)
{
   long lSize = 0;
   ...
   lSize = ftell(f); // ftell returns a long
}
```

Fourth Part : Missing elements

- Initilisation of content
- Liberation of memory
- Closing file

Correction

```
int main(void)
{
   if(buffer != NULL)
   {
      memset(buffer, '\0', lSize + 1);
      ...
      free(buffer);
      buffer = NULL;
   }
   fclose(f);
   f = NULL;
}
```

Software parameters

- Argc, Argv, ...
- Don't forget to verify your parameters (number, size, ...)

```
int main(int argc, char **argv)
{
   if(argc != 2)
      printf("error!");
   dosomething(argv[1]);
   return 0;
}
```

Strings functions

- Strcpy, Strcat, Strlen, Sprintf, Scanf, ... considered as unsecured functions
- Strncpy, Strncat, Strnlen, ... add a third element, the number of elements
- Strcpy_s, Strcat_s, Strlen_s, ... for Visual Studio
- Strncpy_s, Strncat_s, Strnlen_s, ... much more better for Visual Studio
- Do not forget the terminating null cheracter !!!

Printing problem

- Printf, Fprintf, Sprintf, Snprintf, ...
- Overwriting of memory, read memory, ...

```
int main(int argc, char **argv)
{
   char * secret = "This is a secret!\n";

   printf(argv[1]);

   return 0;
}
```

"./program %s" will print the secret on the standard output

Sprintf problem

```
void func(const char *name)
{
    char filename[128];
    sprintf(filename, "%s.txt", name);
}
```

Solution

```
void func(const char *name)
{
   char filename[128];
   sprintf(filename, "%.123s.txt", name);
}

123 characters of name, 4 characters (".txt")
and the null terminator = 128 characters

or

void func(const char *name)
{
   char filename[128];
   snprintf(filename, sizeof(filename), "%s.txt", name);
}
```

Now your turn

- One program who read a file from the desktop
- The path of the file will be passed in parameter of the software
- Copy the content of the file to an other buffer
- Printing the new buffer
- All of this points in a static secure way with error management :)

Dynamic Analysis

- Check the execution of your software.
- Manage memory leaks, threading bugs, ...
- Valgrind example on Linux.
- Visual Studio example on Windows.

Valgrind

- `valgrind -v --leak-check=full --show-reachable=yes ./your_program 2>report.txt`
- Open and parse the file.

Example 1

```
#include <stdio.h>

void f(void)
{
   int *x = malloc(10 * sizeof(int));
   x[10] = 0;
}

int main(void)
{
   f();
   return 0;
}
```

Problems

- Problem 1 : heap block overrun
- problem 2 : memory leak – x not freed

Example 1

Most error messages look like the following, which describes problem 1, the heap block overrun:

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Memory leak messages look like this:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

Example 2

```
#include <stdio.h>
int main(void)
{
   char *p;

   // Allocation #1 of 19 bytes
   p = (char*)malloc(19);

   // Allocation #2 of 12 bytes
   p = (char*)malloc(12);
   free(p);

   // Allocation #3 of 16 bytes
   p = (char*)malloc(16);

   return 0;
}
```

Example 2

This outputs a report to the terminal like

```
==9704== Memcheck, a memory error detector for x86-linux.
==9704== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==9704== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==9704== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==9704== For more details, rerun with: -v
==9704==
==9704==
==9704== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==9704== malloc/free: in use at exit: 35 bytes in 2 blocks.
==9704== malloc/free: 3 allocs, 1 frees, 47 bytes allocated.
==9704== For counts of detected errors, rerun with: -v
==9704== searching for pointers to 2 not-freed blocks.
==9704== checked 1420940 bytes.
==9704==
==9704== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x80483BF: main (test.c:15)
==9704==
==9704==
==9704== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x8048391: main (test.c:8)
==9704==
==9704== LEAK SUMMARY:
==9704==    definitely lost: 35 bytes in 2 blocks.
==9704==    possibly lost:   0 bytes in 0 blocks.
==9704==    still reachable: 0 bytes in 0 blocks.
==9704==         suppressed: 0 bytes in 0 blocks.
```

Explanation

- Allocation #1 (19 byte leak) is lost because p is pointed elsewhere before the memory from Allocation #1 is free'd. In the 19 byte leak entry, the bytes were allocate in test.c, line 8.
- Allocation #2 (12 byte leak) doesn't show up in the list because it is free'd.
- Allocation #3 shows up in the list even though there is still a reference to it (p) at program termination. This is still a memory leak. Valgrind tells us where to look for the allocation (test.c line 15).

Use of uninitialized memory

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

int main(void)
{
   char *p;
   char c = *p;

   printf("\n [%c]\n", c);
   return 0;
}
```

Use of uninitialized memory

```
$ valgrind --tool=memcheck ./val
==2862== Memcheck, a memory error detector
==2862== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2862== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright inf
==2862== Command: ./val
==2862==
==2862== Use of uninitialised value of size 8
==2862==    at 0x400530: main (valgrind.c:8)
==2862==

[#]
==2862==
==2862== HEAP SUMMARY:
==2862==     in use at exit: 0 bytes in 0 blocks
==2862==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2862==
==2862== All heap blocks were freed -- no leaks are possible
==2862==
==2862== For counts of detected and suppressed errors, rerun with: -v
==2862== Use --track-origins=yes to see where uninitialized values come from
==2862== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

As seen from the output above, Valgrind detects the uninitialized variable and gives a warning(see the lines in bold above).

Reading/writing memory after is has been freed

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

int main(void)
{
   char *p = malloc(1);
   *p = 'a';

   char c = *p;

   printf("\n [%c]\n", c);

   free(p);
   c = *p;
   return 0;
}
```

Reading/writing memory after is has been freed

```
$ valgrind --tool=memcheck ./val
==2849== Memcheck, a memory error detector
==2849== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2849== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright inf
==2849== Command: ./val
==2849==

 [a]
==2849== Invalid read of size 1
==2849==    at 0x400603: main (valgrind.c:30)
==2849==  Address 0x51b0040 is 0 bytes inside a block of size 1 free'd
==2849==    at 0x4C270BD: free (vg_replace_malloc.c:366)
==2849==    by 0x4005FE: main (valgrind.c:29)
==2849==
==2849==
==2849== HEAP SUMMARY:
==2849==     in use at exit: 0 bytes in 0 blocks
==2849==   total heap usage: 1 allocs, 1 frees, 1 bytes allocated
==2849==
==2849== All heap blocks were freed -- no leaks are possible
==2849==
==2849== For counts of detected and suppressed errors, rerun with: -v
==2849== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

As seen in the output above, the tool detects the invalid read and prints the warning 'Invalid read of size 1'.

Reading/writing off the end of malloc'd blocks

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

int main(void)
{
   char *p = malloc(1);
   *p = 'a';

   char c = *(p+1);

   printf("\n [%c]\n", c);

   free(p);
   return 0;
}
```

Reading/writing off the end of malloc'd blocks

```
$ valgrind --tool=memcheck ./val
==2835== Memcheck, a memory error detector
==2835== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2835== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright inf
==2835== Command: ./val
==2835==
==2835== Invalid read of size 1
==2835==    at 0x4005D9: main (valgrind.c:25)
==2835==  Address 0x51b0041 is 0 bytes after a block of size 1 alloc'd
==2835==    at 0x4C274A8: malloc (vg_replace_malloc.c:236)
==2835==    by 0x4005C5: main (valgrind.c:22)
==2835==

 []
==2835==
==2835== HEAP SUMMARY:
==2835==     in use at exit: 0 bytes in 0 blocks
==2835==   total heap usage: 1 allocs, 1 frees, 1 bytes allocated
==2835==
==2835== All heap blocks were freed -- no leaks are possible
==2835==
==2835== For counts of detected and suppressed errors, rerun with: -v
==2835== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Again, this tool detects the invalid read done in this case.

Memory leaks

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

int main(void)
{
   char *p = malloc(1);
   *p = 'a';

   char c = *p;

   printf("\n [%c]\n", c);

   return 0;
}
```

Memory leaks

```
$ valgrind --tool=memcheck --leak-check=full ./val
==2888== Memcheck, a memory error detector
==2888== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2888== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright inf
==2888== Command: ./val
==2888==

 [a]
==2888==
==2888== HEAP SUMMARY:
==2888==     in use at exit: 1 bytes in 1 blocks
==2888==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==2888==
==2888== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2888==    at 0x4C274A8: malloc (vg_replace_malloc.c:236)
==2888==    by 0x400575: main (valgrind.c:6)
==2888==
==2888== LEAK SUMMARY:
==2888==    definitely lost: 1 bytes in 1 blocks
==2888==    indirectly lost: 0 bytes in 0 blocks
==2888==      possibly lost: 0 bytes in 0 blocks
==2888==    still reachable: 0 bytes in 0 blocks
==2888==         suppressed: 0 bytes in 0 blocks
==2888==
==2888== For counts of detected and suppressed errors, rerun with: -v
==2888== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

The lines (in bold above) shows that this tool was able to detect the leaked memory.

Mismatched use of malloc/new/new[] vs free/delete/delete[]

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

int main(void)
{
   char *p = malloc(1);
   *p = 'a';

   char c = *p;

   printf("\n [%c]\n", c);
   delete p;
   return 0;
}
```

Mismatched use of malloc/new/new[] vs free/delete/delete[]

```
$ valgrind --tool=memcheck --leak-check=full ./val
==2972== Memcheck, a memory error detector
==2972== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==2972== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright inf
==2972== Command: ./val
==2972==

 [a]
==2972== Mismatched free() / delete / delete []
==2972==    at 0x4C26DCF: operator delete(void*) (vg_replace_malloc.c:387)
==2972==    by 0x40080B: main (valgrind.c:13)
==2972==  Address 0x595e040 is 0 bytes inside a block of size 1 alloc'd
==2972==    at 0x4C274A8: malloc (vg_replace_malloc.c:236)
==2972==    by 0x4007D5: main (valgrind.c:7)
==2972==
==2972==
==2972== HEAP SUMMARY:
==2972==     in use at exit: 0 bytes in 0 blocks
==2972==   total heap usage: 1 allocs, 1 frees, 1 bytes allocated
==2972==
==2972== All heap blocks were freed -- no leaks are possible
==2972==
==2972== For counts of detected and suppressed errors, rerun with: -v
==2972== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

We see from the output above (see lines in bold), the tool clearly states 'Mismatched free() / delete / delete []'

Doubly freed memory

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

int main(void)
{
   char *p = malloc(1);
   *p = 'a';

   char c = *p;

   printf("\n [%c]\n", c);
   free(p);
   free(p);
   return 0;
}
```

Doubly freed memory

```
$ valgrind --tool=memcheck --leak-check=full ./val
==3167== Memcheck, a memory error detector
==3167== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==3167== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright inf
==3167== Command: ./val
==3167==

 [a]
==3167== Invalid free() / delete / delete[]
==3167==    at 0x4C270BD: free (vg_replace_malloc.c:366)
==3167==    by 0x40060A: main (valgrind.c:12)
==3167==  Address 0x51b0040 is 0 bytes inside a block of size 1 free'd
==3167==    at 0x4C270BD: free (vg_replace_malloc.c:366)
==3167==    by 0x4005FE: main (valgrind.c:11)
==3167==
==3167==
==3167== HEAP SUMMARY:
==3167==     in use at exit: 0 bytes in 0 blocks
==3167==   total heap usage: 1 allocs, 2 frees, 1 bytes allocated
==3167==
==3167== All heap blocks were freed -- no leaks are possible
==3167==
==3167== For counts of detected and suppressed errors, rerun with: -v
==3167== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

As seen from the output above(lines in bold), the tool detects that we have called free twice on the same pointer.

Visual Studio

- Only in C native, no C#, no Visual Basic, no Web.
- http://msdn.microsoft.com/en-us/library/e5ewb1h3(v=vs.90).aspx

Exercice

- Take the source code inside the file named DynStat.c.
- Use Visual Studio arguments / Valgrind / Flawfinder / CPPCheck and your mind/knowledge to fix and make the source code clean in the static and dynamic way.
- Use and abuse of the MSDN.