

Secure programming



Travaux pratiques n ° 2 : Format String attacks

Antoine Puissant

Enseignant : M. Dechaux (ESIEA)

2014 - 2015

Résumé

L'objectif de ce TP est d'essayer différents formats de chaînes de caractères sur trois systèmes d'exploitation différents.

Table des matières

1	XP Virtual Machine and Windbg	3
1.1	Memory leak with Printf	3
1.2	Understand the difference	3
1.3	Crashing your program	3
1.4	Viewing the stack	5
1.5	Overwriting the memory	6
2	Linux and GCC	8
2.1	Crashing your program	8
2.2	Viewing the stack	9
2.3	Overwriting the memory	10
3	Windows 7 and Visual Studio	11
	Table des figures	12

1 XP Virtual Machine and Windbg

1.1 Memory leak with Printf

Nous avons ici deux codes :

```
#include <stdio.h>
void main(int argc, char **argv)
{
    printf(argv[1]);
}
```

Code 1 – Premier programme

```
#include <stdio.h>
void main(int argc, char **argv)
{
    printf("%s\n", argv[1]);
}
```

Code 2 – Second programme

1.2 Understand the difference

Ces deux codes, une fois compilés, fonctionnent tous les deux de la même manière :

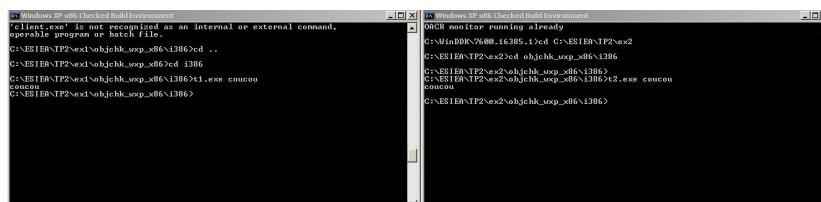


FIGURE 1 – Résultat de l'exécution des deux programmes

Sur la figure 1, il est possible de voir que ces deux programmes réagissent de la même façon avec l'argument qu'il leur est fourni.

1.3 Crashing your program

En exécutant le premier programme avec « %s » en paramètre, nous obtenons le résultat suivant :

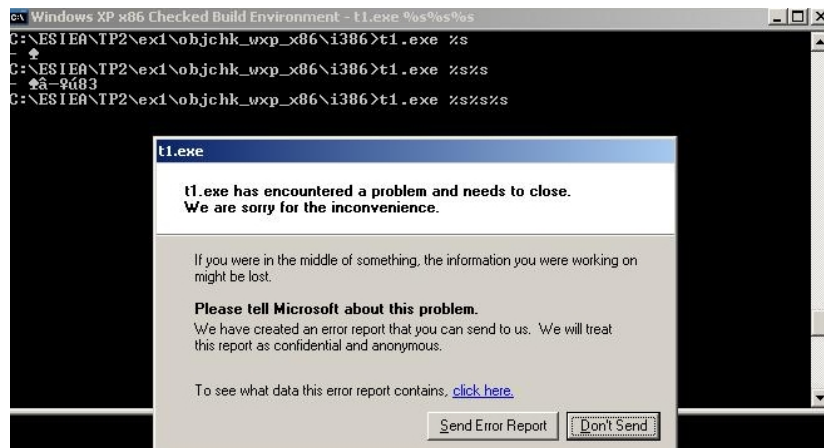


FIGURE 2 – Résultat de l'exécution du premier programme avec « %s » en paramètre

Comme on peut le voir sur la figure 2, il faut trois fois l'argument « %s » pour que le programme crash.

Le programme va crasher car il va essayer de lire une variable qui n'a jamais été instanciée. De ce fait, lors de l'utilisation de deux « %s » dans l'argument, il va accéder à l'espace mémoire dédié au programme. A partir du troisième, il sort de l'espace mémoire alloué et va alors essayer de lire une zone dont l'accès lui est interdit.

Cela peut se vérifier en utilisant l'outil *WinDBG*.

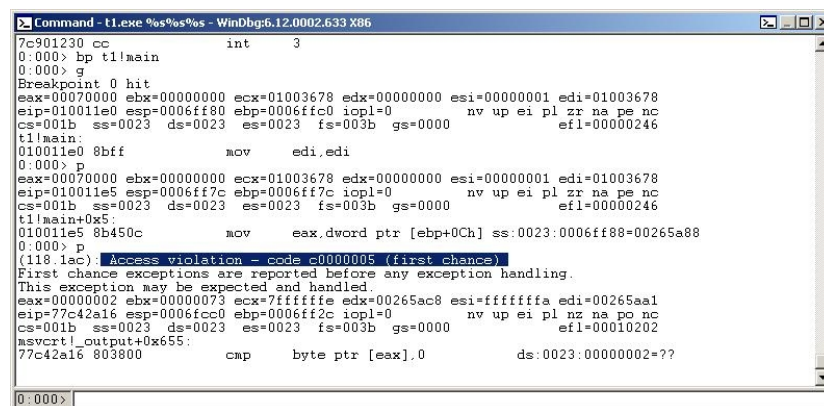


FIGURE 3 – Résultat de l'exécution du premier programme avec *WinDBG*

Comme on peut le voir sur la figure 3, *WinDBG* nous signale que le programme rencontre une erreur suite à une tentative d'accès sur une zone mémoire non autorisée :

```
(118.1ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000002 ebx=00000073 ecx=7fffffff edx=00265ac8 esi=fffffffa edi=00265aa1
eip=77c42a16 esp=0006fccc ebp=0006ff2c iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
msvcrt!_output+0x655:
77c42a16 803800          cmp     byte ptr [eax],0          ds:0023:00000002=??
```

1.4 Viewing the stack

Si l'on donne cette fois en argument à notre premier programme la chaîne de caractères « %08x », on obtient le résultat suivant :

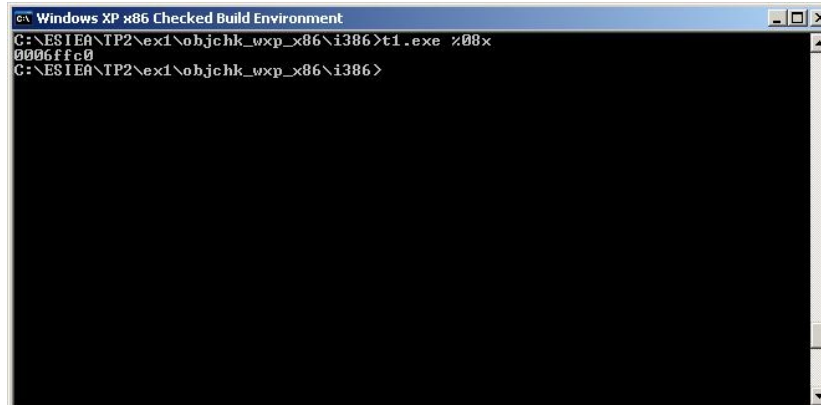


FIGURE 4 – Résultat de l'exécution du premier programme avec « %08x » en paramètre

On se rend alors compte que nous avons ici une adresse mémoire, plus précisément l'adresse 06FFC0. En effet, cette commande (« %x ») va permettre d'afficher une donnée en hexadécimal. Ici, la donnée n'étant pas précisée, elle va donc afficher l'espace mémoire de l'appel.

En donnant en argument « %08x%08x », il est possible de retrouver deux valeurs intéressantes du programme. En effet, dans ce dernier il n'y a aucune variable de stockée en mémoire. Ainsi, la première adresse mémoire sur laquelle notre programme va lire avec « %08x » sera celle d'*EBP*. Cela se vérifie avec l'outil *WinDBG* :

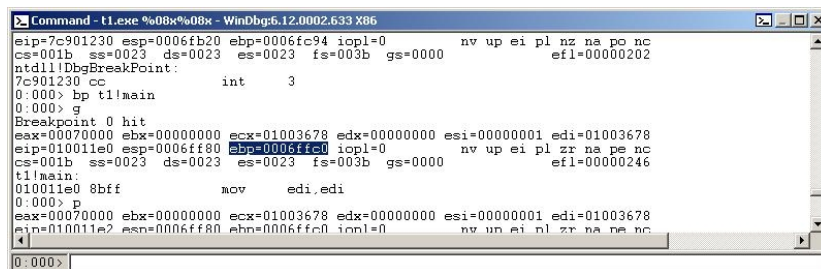


FIGURE 5 – État de la mémoire au début de l'exécution du programme

Comme on peut le voir sur la figure 5, *EBP* est stocké à l'adresse mémoire 0006FFC0. Or comme le montre la figure 6, à l'exécution, nous retrouvons cette première adresse sur le prompt du programme.

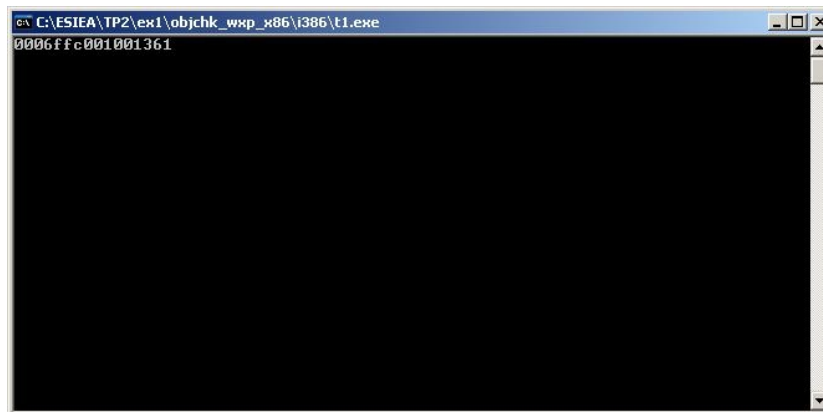


FIGURE 6 – Résultat de l'exécution du troisième programme

La seconde adresse correspond quand à elle à l'adresse de *EIP*. En effet, nous savons que l'adresse d'*EIP* se situe 4 octets après celle d'*EBP* :

$$EIP = EBP + 4 = 06FFC0 + 4 = 06FFC4 \quad (1)$$

1.5 Overwriting the memory

Nous avons le code numéro trois suivant :

```
#include <stdio.h>
void main(int argc, char **argv){
    int bytes;
    printf("%s\n\n", argv[1], &bytes);
    printf("You input %d characters\n", bytes);
}
```

Code 3 – Troisième programme

En exécutant ce code avec la chaîne de caractères *coucou*, nous obtenons le résultat suivant :

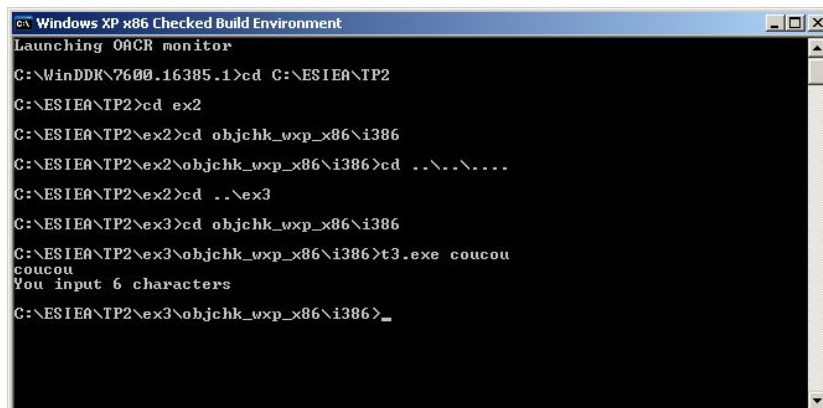


FIGURE 7 – Résultat de l'exécution du troisième programme

Ici, le programme nous retourne la chaîne donnée en paramètre puis affiche sur une nouvelle ligne :

You input 6 characters

Cette phrase va varier en fonction du nombre caractères donnés en argument. Cela est fait grâce à l'argument `%n` de la fonction `printf`. En effet, ce dernier va permettre de stocker le nombre de caractères déjà affiché par la fonction `printf`. Ainsi, en exécutant ce programme avec `WinDBG`, il est possible d'observer cela. En effet, en donnant comme argument au programme une chaîne de caractères composée de 24 lettres, nous pouvons observer que le compte est réalisé correctement.

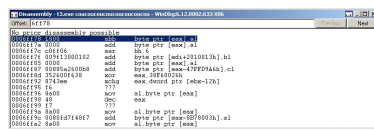


FIGURE 8 – Fenêtre *Disassembly*
après le premier *printf* du
troisième programme

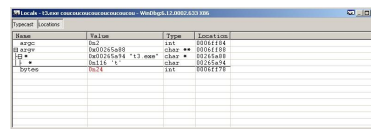


FIGURE 9 – Fenêtre *Locals* après le premier *printf* du troisième programme

Sur la figure 8, nous pouvons lire que l'adresse mémoire `06FF78(EBP - 4)` dans laquelle est stockée la variable `bytes` a pour valeur `0x18`¹. Sur la figure 9, on va aussi pouvoir retrouver la valeur de la variable, cette fois égale à `0n24`. Ainsi, on peut confirmer que la taille de l'argument a bien été comptée.

1. 24 en décimal


```

blackyfox@blackyfox-EasyNote-LM86:~/Bureau/tp2$ valgrind ./exo1 %s%s%s%s%s%s
==21908== Memcheck, a memory error detector
==21908== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==21908== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==21908== Command: ./exo1 %s%s%s%s%s%s
==21908==
==21908== Conditional jump or move depends on uninitialised value(s)
==21908==    at 0x4E827AB: vfprintf (vfprintf.c:1661)
==21908==    by 0x4E8B498: printf (printf.c:33)
==21908==    by 0x400553: main (in /home/blackyfox/Bureau/tp2/exo1)
==21908==
==21908== Use of uninitialised value of size 8
==21908==    at 0x4E82A03: vfprintf (vfprintf.c:1661)
==21908==    by 0x4E8B498: printf (printf.c:33)
==21908==    by 0x400553: main (in /home/blackyfox/Bureau/tp2/exo1)
==21908==
==21908== Invalid read of size 1
==21908==    at 0x4E82A03: vfprintf (vfprintf.c:1661)
==21908==    by 0x4E8B498: printf (printf.c:33)
==21908==    by 0x400553: main (in /home/blackyfox/Bureau/tp2/exo1)
==21908== Address 0x200000000 is not stack'd, malloc'd or (recently) free'd
==21908==
==21908== Process terminating with default action of signal 11 (SIGSEGV)
==21908== Access not within mapped region at address 0x200000000
==21908==    at 0x4E82A03: vfprintf (vfprintf.c:1661)
==21908==    by 0x4E8B498: printf (printf.c:33)
==21908==    by 0x400553: main (in /home/blackyfox/Bureau/tp2/exo1)
==21908== If you believe this happened as a result of a stack
==21908== overflow in your program's main thread (unlikely but
==21908== possible), you can try to increase the size of the
==21908== main thread stack using the --main-stacksize= flag.
==21908== The main thread stack size used in this run was 8388608.
==21908==
==21908== HEAP SUMMARY:
==21908==    in use at exit: 0 bytes in 0 blocks
==21908==    total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==21908==
==21908== All heap blocks were freed -- no leaks are possible
==21908==
==21908== For counts of detected and suppressed errors, rerun with: -v
==21908== Use --track-origins=yes to see where uninitialised values come from
==21908== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
Erreur de segmentation (core dumped)
blackyfox@blackyfox-EasyNote-LM86:~/Bureau/tp2$

```

FIGURE 11 – Résultat de l'exécution du premier programme avec *Valgrind*

Ainsi, comme précédemment, nous pouvons remarquer que comme vu dans le point 1.3 que le programme essayer d'accéder à une zone mémoire qui lui est interdite. Ainsi, le programme ne pouvant pas lire la donnée en mémoire s'arrête.

2.2 Viewing the stack

Tout comme pour le point 1.4, nous allons ici retrouver des adresses mémoire en donnant en argument « %08x » à notre premier programme :

```

blackyfox@blackyfox-EasyNote-LM86:~/Bureau/tp2$ ./exo1 "%08x %08x %08x"
38830a58 38830a70 00000000blackyfox@blackyfox-EasyNote-LM86:~/Bureau/tp2$

```

FIGURE 12 – Résultat de l'exécution du premier programme avec « %08x » en argument

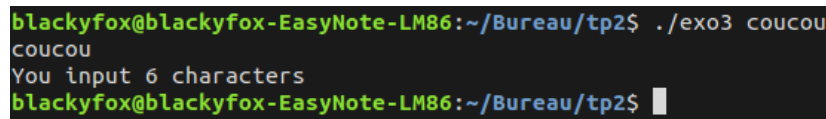
Le premier « %08x » va nous permettre de retrouver l'adresse mémoire d'*EBP*, ici égale à 38830A56.

Cependant, nous ne retrouvons pas la valeur de EIP dans le second argument. En effet, nous savons que $EIP = EBP + 4$. Ainsi, nous devrions retrouver l'adresse 38830A5A en seconde adresse de sortie. Nous avons cependant ici l'adresse 38830A70.

Nous pouvons donc conclure que nous le compilateur GCC n'agit pas de la même façon que dans notre test précédent, sur la machine virtuelle de Windows XP.

2.3 Overwriting the memory

Ici encore, nous pouvons retrouver le même type de sortie que sur la machine virtuelle de Windows XP (c.f. 1.5) :



```
blackkyfox@blackkyfox-EasyNote-LM86:~/Bureau/tp2$ ./exo3 coucou
coucou
You input 6 characters
blackkyfox@blackkyfox-EasyNote-LM86:~/Bureau/tp2$
```

FIGURE 13 – Résultat de l'exécution du troisième programme

Ainsi, la fonction *printf* va ici aussi compter le nombre de caractères écrit puis stocker cette valeur en mémoire.

Il est possible de vérifier cela en utilisant *Valgrind* :

```

==22618== TO CONTROL THIS PROCESS USING vgdb (which you probably
==22618== don't want to do, unless you know exactly what you're doing,
==22618== or are doing some strange experiment):
==22618== /usr/lib/valgrind/../../bin/vgdb --pid=22618 ...command...
==22618==
==22618== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==22618== /path/to/gdb ./exo3
==22618== and then give GDB the following command
==22618== target remote | /usr/lib/valgrind/../../bin/vgdb --pid=22618
==22618== --pid is optional if only one valgrind process is running
==22618==
--22618-- REDIR: 0x4019ca0 (strlen) redirected to 0x38068331 (???)
--22618-- Reading syms from /usr/lib/valgrind/vgpreload_core-amd64-linux.so
--22618-- Considering /usr/lib/valgrind/vgpreload_core-amd64-linux.so ..
--22618-- .. CRC mismatch (computed 329d6860 wanted c0186920)
--22618-- object doesn't have a symbol table
--22618-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
--22618-- Considering /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so ..
--22618-- .. CRC mismatch (computed 1fb85af8 wanted 2e9e3c16)
--22618-- object doesn't have a symbol table
==22618== WARNING: new redirection conflicts with existing -- ignoring it
--22618-- old: 0x04019ca0 (strlen) ) R-> (0000.0) 0x38068331 ???
--22618-- new: 0x04019ca0 (strlen) ) R-> (2007.0) 0x04c2e1a0 strlen
--22618-- REDIR: 0x4019a50 (index) redirected to 0x4c2dd50 (index)
--22618-- REDIR: 0x4019c70 (strcmp) redirected to 0x4c2f2f0 (strcmp)
--22618-- REDIR: 0x401a9c0 (memcpy) redirected to 0x4c31da0 (memcpy)
--22618-- Reading syms from /lib/x86_64-linux-gnu/libc-2.19.so
--22618-- Considering /lib/x86_64-linux-gnu/libc-2.19.so ..
--22618-- .. CRC mismatch (computed dc620abc wanted 148cbd6e)
--22618-- Considering /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.19.so ..
--22618-- .. CRC is valid
--22618-- REDIR: 0x4ec3d60 (strcascmp) redirected to 0x4a25720 (_vgnU_ifunc_wrapper)
--22618-- REDIR: 0x4ec6050 (strncasecmp) redirected to 0x4a25720 (_vgnU_ifunc_wrapper)
--22618-- REDIR: 0x4ec3530 (memcpy@GLIBC_2.2.5) redirected to 0x4a25720 (_vgnU_ifunc_wrapper)
--22618-- REDIR: 0x4ec17c0 (rindex) redirected to 0x4c2da30 (rindex)
--22618-- REDIR: 0x4ecaac0 (strchrnul) redirected to 0x4c319b0 (strchrnul)
coucou
You input 6 characters
--22618-- REDIR: 0x4eb9df0 (free) redirected to 0x4c2bd80 (free)
==22618==
==22618== HEAP SUMMARY:
==22618== in use at exit: 0 bytes in 0 blocks
==22618== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==22618==
==22618== All heap blocks were freed -- no leaks are possible
==22618==
==22618== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==22618== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
blackyfox@blackyfox-EasyNote-LM86:~/Bureau/tp2$

```

FIGURE 14 – Résultat de l'exécution du troisième programme

Comme le montre la figure 14, nous pouvons remarquer que le programme utilise la fonction *strlen*. C'est en effet cette dernière qui va permettre à la fonction *printf* de connaître la taille des données écrites. On va ensuite retrouver l'utilisation de la fonction *memcpy* qui va permettre au programme de copier la valeur comptée dans la variable *byte* du programme.

3 Windows 7 and Visual Studio

Table des figures

1	Résultat de l'exécution des deux programmes	3
2	Résultat de l'exécution du premier programme avec « %s » en paramètre	4
3	Résultat de l'exécution du premier programme avec <i>WinDBG</i> . .	4
4	Résultat de l'exécution du premier programme avec « %08x » en paramètre	5
5	État de la mémoire au début de l'exécution du programme . . .	5
6	Résultat de l'exécution du troisième programme	6
7	Résultat de l'exécution du troisième programme	6
8	Fenêtre <i>Disassembly</i> après le premier <i>printf</i> du troisième programme	7
9	Fenêtre <i>Locals</i> après le premier <i>printf</i> du troisième programme .	7
10	Résultat de l'exécution du premier programme	8
11	Résultat de l'exécution du premier programme avec <i>Valgrind</i> . .	9
12	Résultat de l'exécution du premier programme avec « %08x » en argument	9
13	Résultat de l'exécution du troisième programme	10
14	Résultat de l'exécution du troisième programme	11