Práctica 2.1

EJERCICIO DE DESARROLLO SEGURO – SECURE CODE

José Ángel Dorado González

Seguridad Informática

ÍNDICE

1º)¿Qué es "Morris Worm"?¿Qué, dónde, cómo, cuándo y por qué?

El gusano Morris fue el primer ejemplar de malware de autorreplicación que afectó a internet (en aquel entonces ARPANET), fue creado por Robert Tappan Moris el 2 de noviembre de 1988, afectando aproximadamente a 6000 de los 60.000 servidores conectados a la red.

Su funcionamiento se basaba en el desbordamiento de búfer, provocado cuando se permite que el usuario ingrese más datos de los que esperaba su programa, esto permite modificaciones arbitrarias en la memoria. Debido a la forma en que la pila está configurada, el atacante puede escribir código en la memoria. El gusano Morris era capaz de ralentizar e incluso dañar los ordenadores que afectaba.

Según declara el autor, creó un programa con gran capacidad de reproducirse pero nunca pensó que se propagaría tan rápido y tan extensamente. Morris afirmó que se cometió un error al propagar el gusano, convirtiéndose así en el descubridor de uno de los componentes más importantes y peligrosos de los malware de la actualidad.

2º)¿Qué es un Buffer Overflow Attack?. Explica y escribe un breve ejemplo.

Como se comentó anteriormente, esto ocurre cuando permite que el usuario ingrese más datos de los que esperaba su programa, lo que permite modificaciones arbitrarias, permitiendo que un atacante pueda escribir código en dicha memoria.

Por ejemplo, si un programa tiene definidos dos elementos de datos continuos en memoria: un buffer de 8 bytes tipo string A y otro de dos bytes tipo entero B. Al principio, el buffer A es completamente nulo y B contiene el número 3, quedaría representado así:

0	0	0	0	0	0	0	0	0	3
Buffer A								Buffer B	

Ahora si intentamos almacenar la cadena “demasiado” en el buffer A, al ser este buffer de menor tamaño que la cadena, se sobrescribe el valor que contenía B:

‘d’	‘e’	‘m’	‘a’	‘s’	‘i’	‘a’	‘d’	‘o’	0
Buffer A								Buffer B	

A pesar de que el programador no quería cambiar el contenido del búfer B, el valor de éste ha sido reemplazado por un número equivalente a parte de la cadena de caracteres.

Esto en código fuente se puede ver tal que así:

Declaramos una matriz de 100 bytes:

char memory[100];

Nosotros podemos realizar lo siguiente:

memory[150] = 'a';

No hay verificaciones de límites en la matriz y el código puede incluso funcionar, pero no se debe jugar con la memoria que no has pedido, por lo que probablemente no haga las cosas así. Lo que sucede, como se explicó en el ejemplo de antes, es que la función a la que llama sobrescribirá la memoria.

3º)¿Qué es un Double Free Attack?. Explica y escribe un breve ejemplo.

Este es un ataque más sofisticado que afecta a algunas implementaciones de malloc. El ataque puede ocurrir cuando se llama a un puntero que ya se ha liberado antes de que se haya reinicializado con una nueva dirección de memoria:

```
free(x);
/* código */
free(x);
```

Esto nunca debería ocurrir en el código, y la forma más fácil de evitarlo es simplemente configurar el puntero para que apunte a NULL una vez se haya liberado:

```
free(x);
x = NULL;
/* código */
free(x);
```

Este error es más difícil de explotar que los desbordamientos de búfer, sin embargo es importante asegurarse de que el código libere correctamente los bloques de memoria válidos.

4º)Explica y comenta qué errores tiene o qué problemas nos podemos encontrar con cada uno de los siguientes ejemplos en Lenguaje C:

a)

```
int stupid(int a) {
        return (a+1) > a;
}
```

Existe un error en esta función, y es que quiere devolver un valor con return pero ahí no se devuelve nada, ya que lo que está haciendo es una comparación absurda.

b)

```
int *i = (int*)malloc(sizeof(int));
int j = 0;
*i = 4;
free(i);
/* ...el programa continúa */
*i = j;
```

Aquí existe un error basado en que cuando se libera la memoria de un puntero con la función free, este queda apuntando a una dirección que no es correcta, debido a que free() no lo hace apuntar a NULL automáticamente. Posteriormente se le asigna al puntero ya liberado, el valor de j, esto es erróneo ya que la memoria solicitada fue liberada, todo esto causará problemas.

c)

```
char buf[N];
char buf2[N-1]
char *q = buf;
char *p = buf2;
while (*p)
 *q++ = *p++;
```

Aquí existe un error provocado por un desbordamiento de buffer, debido a que la variable p es más pequeña que q.

d)

```
//creating integer of size n.
int *piBuffer = malloc(n * sizeof(int));

//Assigned value to allocated memory
for (i = 0; i < n; ++i)
{
        piBuffer [i] = i * 3;
}
```

Aquí ocurre un problema debido a que al reservar la memoria, no se comprueba si hay memoria libre o no.

5º) ¿Qué quieren decir los siguientes consejos básicos?

- "Don’t ignore compiler warnings"
    - Este consejo nos especifica que no debemos ignorar los warnings que pueden aparecernos en el compilador a la hora de compilar, estas advertencias son mensajes que muestra el compilador debido a que se ha detectado una anomalía, aunque como no es un error crítico, se asumen ciertas condiciones y la traducción del programa continúa. Esto puede provocar que aunque no se detenga la compilación del programa, este no se ejecute tal y como se había programado, provocando fallos.
- Don’t write complex code
    - Este consejo nos intenta transmitir que no hay que crear un código complejo porque esto es una mala práctica. Aunque si se está escribiendo un algoritmo matemático, quizás el código si es complejo, pero a lo que se refiere es en no transformar un código que debería ser simple, en difícil de entender porque lo adornamos nosotros.
- Usa enum para los códigos de error
    - Este consejo nos dice que debemos usar enum para los códigos de error, esto es debido a que los errores son un conjunto limitado de variables por lo que hay que considerar la posibilidad de usar dicha enumeración. Con esto conseguimos que el código sea más claro y legible, además, las enumeraciones nos ofrecen una manera más sencilla de trabajar con conjuntos de constantes relacionadas.
- Usa Fixed-width Data Types: You should use fixed length data type (uint8_t,uint16_t ...etc) in place of implementation defined (int,long, ...etc). In C99, C committee introduce <stdint.h> that define fixed length data types.
    - Este consejo nos dice que debemos utilizar Fixed-width Data Types, esto es debido a que por ejemplo int8_t o int32_t tienen un tamaño específico, mientras que por ejemplo int puede ser de cualquier tamaño mayor o igual a 16 bits. Siempre hay que tener en cuenta el tamaño de la variable, porque si nosotros declaramos esto: *‘int i = 10’*, en algunos sistemas puede ser un entero de 16 bits, en otros de 32 bits e incluso en los sistemas más nuevos puede ser de 32 bits. Esto puede dar lugar a resultados extraños, por lo que se recomienda especificar variables de tamaño fijo.

6º)Ejecuta el siguiente código y comprueba y comenta el resultado

```
#include <stdio.h>

int main(void)
{

	unsigned int uiData = 2;
	int iData = -20;

	if(iData + uiData > 6)
	{
		printf("%s\n", "a+b > 6");
	}
	else
	{
		printf("%s\n", "a+b < 6");
	}

	return 0;
}
```

Una vez compilado y ejecutado el código, sabiendo que *uiData = 2* y que *iData = -20*, podemos observar que el resultado de la operación *uiData + iData* siempre da el resultado *"a+b > 6"*. Esto en una suma normal sería algo erróneo, pero hay que fijarse que la variable *uiData* es del tipo unsigned int, este tipo sólo permite representar números mayores o iguales a 0. Esto provoca que la variable *iData* que es negativa, al sumarse con uiData, se convierta en positiva, por lo que si se suma 20 + 2 el resultado dará 22, dando como resultado que *a+b* es mayor a 6.

Bibliografía

https://www.cprogramming.com/tutorial/secure.html

https://glooscapsoftware.blogspot.com/2022/02/dont-ignore-compiler-warnings-unless.html

https://www.quora.com/Is-it-a-bad-practice-to-write-complex-and-hard-to-understand-code

https://learn.microsoft.com/es-es/dotnet/visual-basic/programming-guide/language-features/constants-enums/when-to-use-an-enumeration

https://en.cppreference.com/w/cpp/types/integer

https://www.badprog.com/c-type-what-are-uint8-t-uint16-t-uint32-t-and-uint64-t