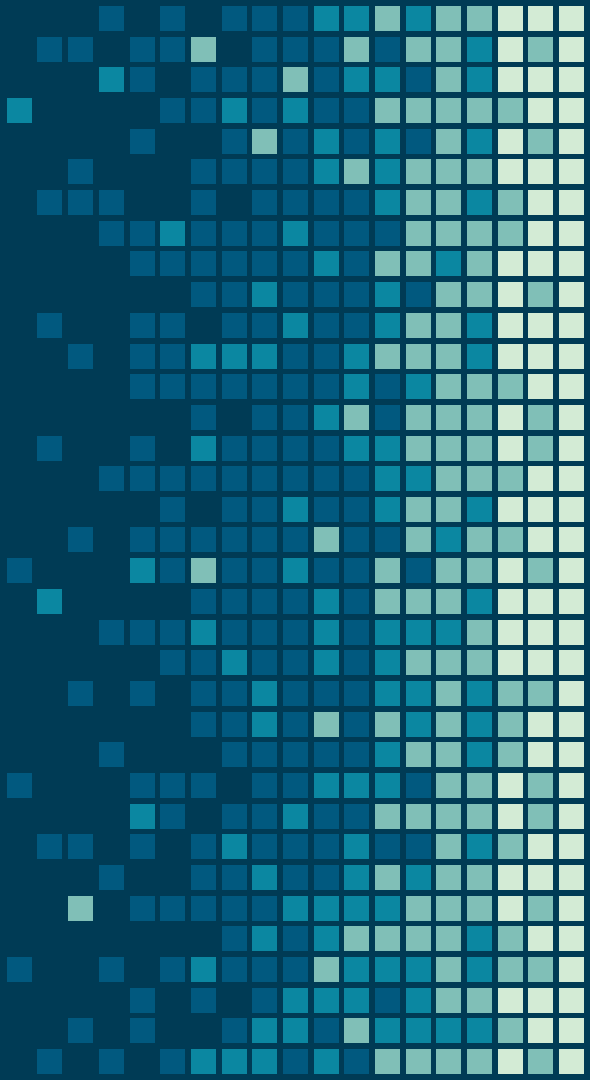
Seguridad



Agenda

- Introducción
- Conceptos sobre memoria
- Demostración

Introducción

Introducción

- La seguridad es una las partes más críticas de los Sistemas Computacionales en general. Dinero, vidas, artefactos dependen de esto.
- Es sencillo caer en la frase “Mis sistema es seguro”, ya que pasó todas las pruebas.
- “Confiar” en otros.
- No pensara como atacante.
- Asumir que la seguridad se puede garantizar.

Introducción

- Para que un programa sea útil debe recibir datos de entrada.
- Los programadores típicamente hacen el código para manipular los datos de entrada de alguna manera.
- Sin embargo, usualmente no se captura todas las desviaciones del comportamiento deseado del software.
- Aunque la lógica sea correcta, la implementación puede variar dependiendo del programador.

Introducción

- En bajo nivel, en qué lenguaje se programa, generalmente?
- Esto es medio peligroso (C y C++).
- Se permite uso de punteros.
- Acceso a memoria.
- Ofrece gran rendimiento.
- Java ofrece seguridad de memoria, pero no lo arregla todo.

Vulnerabilidad y Exploits

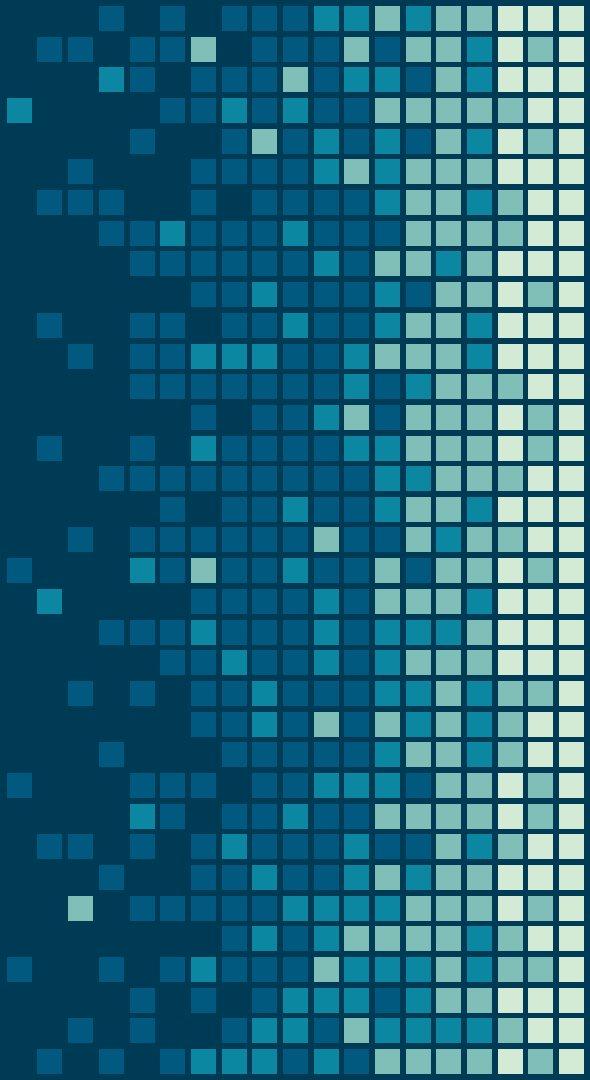
Vulnerabilidad

- Falla del programa que causa que el programa no se comporte como el programador desee y esto permite realizar operaciones indeseadas.

Exploits

- Entrada de dato que cuando se presenta en un programa, activa un evento particular de vulnerabilidad.
- Los atacantes pueden utilizar los exploits para ejecutar operaciones sin autorización.
- Programas vulnerables ejecutan operaciones en algún nivel de privilegio.

Desbordamiento de buffer en C

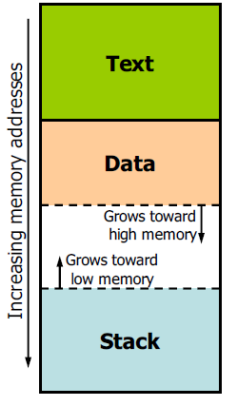


Buffer o arreglos en C

- C, permite aritmética de punteros.
- El compilador no tiene noción del tamaño de un objeto.
- Entonces los programadores deben de verificar de manera explícita el rango de posiciones.
- El desbordamiento se utiliza en muchos exploits.
  - Una entrada muy larga que desborda el arreglo puede ser muy peligrosa.
  - Desbordamiento del control de flujo por el direccionamiento del código (overflow).

Mapa de memoria de un proceso

- Texto: código, instrucciones, datos de solo lectura, tamaño ajustado en tiempo de compilación
- Data: datos inicializados y sin inicializar.
- Stack: LIFO, variables locales, funciones, argumentos, llamadas.

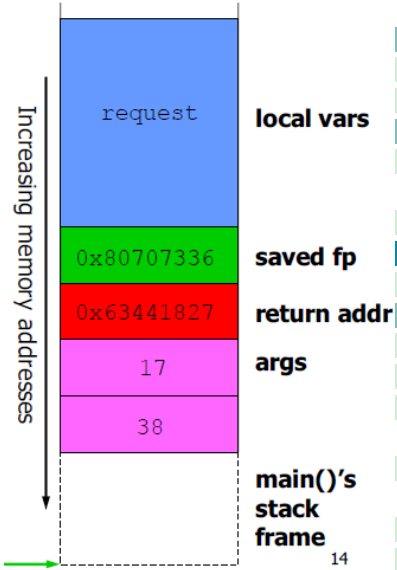


Intel X86 Stack

- Stack frame: Región del stack utilizada para una función en C.
- Variables locales se almacenan en el stack.
- SP: StackPointer.
- BP: frame pointer. Apunta al final del stack frame

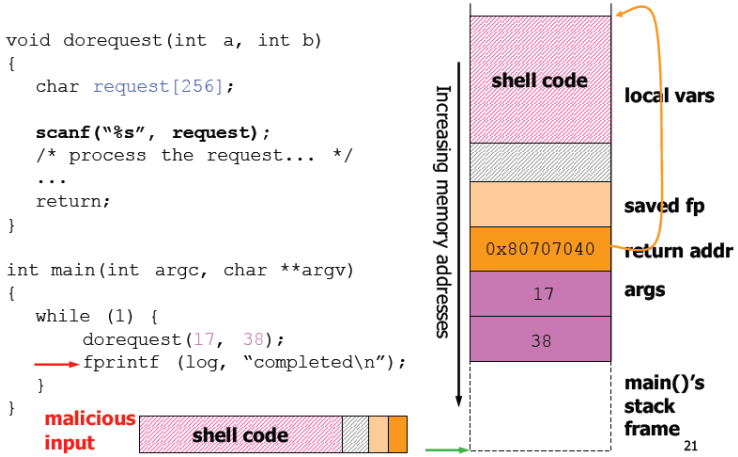
Ejemplo de una función

```
void dorequest(int a, int b)
{
   char request[256];

   scanf("%s", request);
   /* process the request... */
   ...
   return;
}

int main(int argc, char **argv)
{
   while (1) {
        dorequest(17, 38);
   → fprintf (log, "completed\n");
   }
}
```



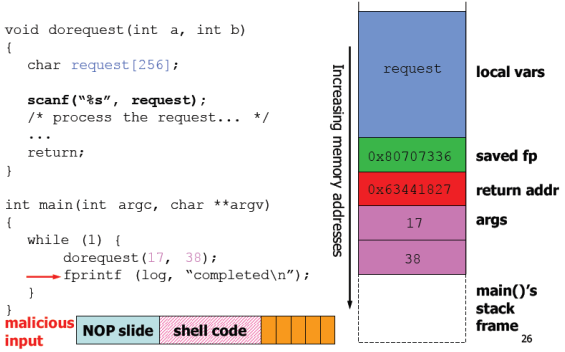
¿Cómo se podría vulnerar el código?

- Utilizando Shell code, muy similar a SQL Injection. Es una entrada maliciosa.



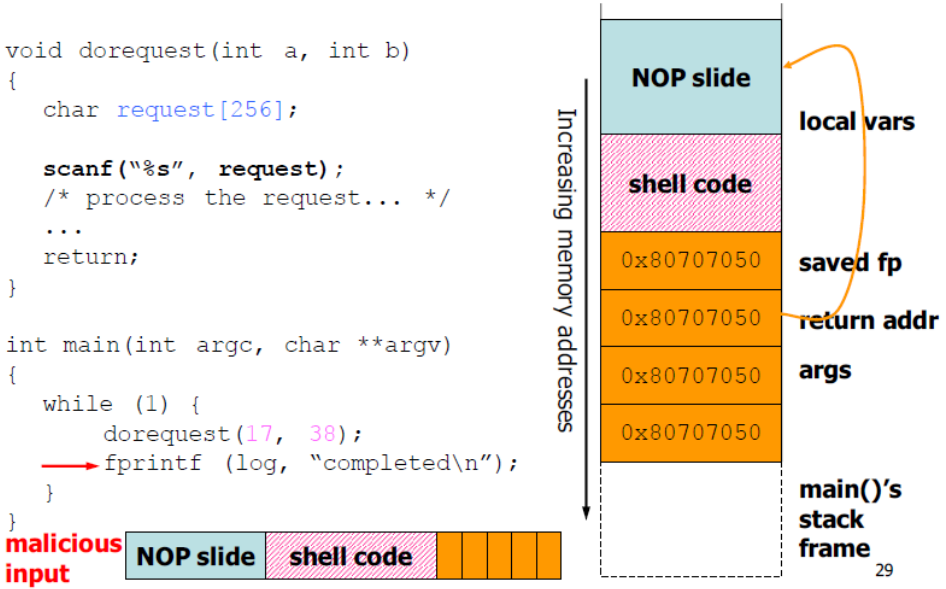
Diseñando un Exploit en el stack

- Parece ser que se necesita las direcciones exactas para saltar, por ejemplo, el del valor del retorno.
- Sin embargo, se puede utilizar NOPS. La idea es brincar a un espacio de NOPS y luego ejecutar el código.
- Entonces se podría repetir direcciones muchas veces hasta que calce donde sirva.

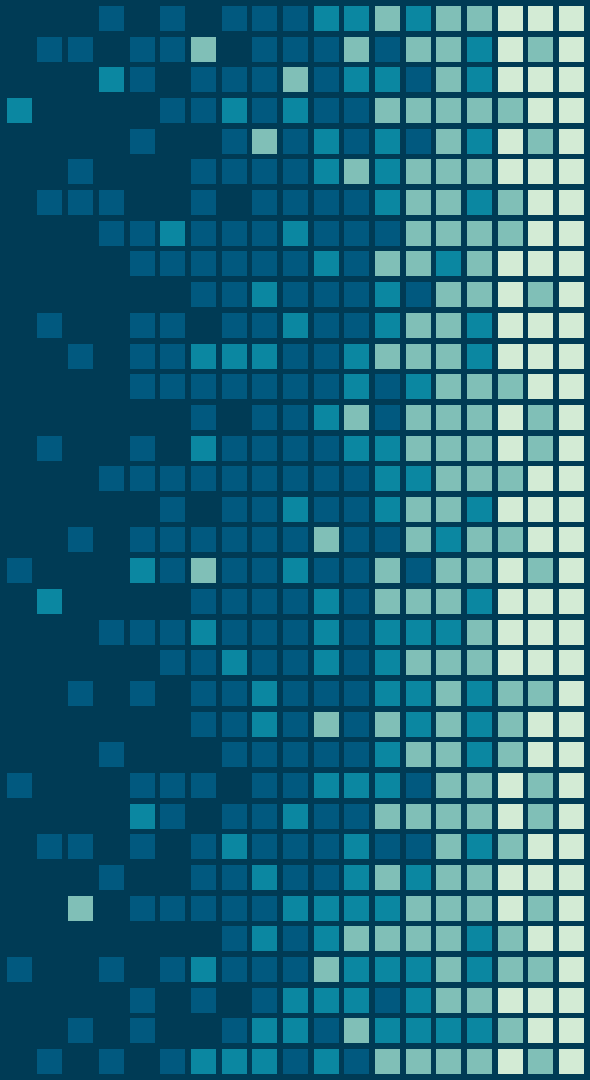


Diseñando un Exploit en el stack

```
void dorequest(int a, int b)
{
   char request[256];

   scanf("%s", request);
   /* process the request... */
   ...
   return;
}

int main(int argc, char **argv)
{
   while (1) {
        dorequest(17, 38);
   →    fprintf (log, "completed\n");
   }
}
```



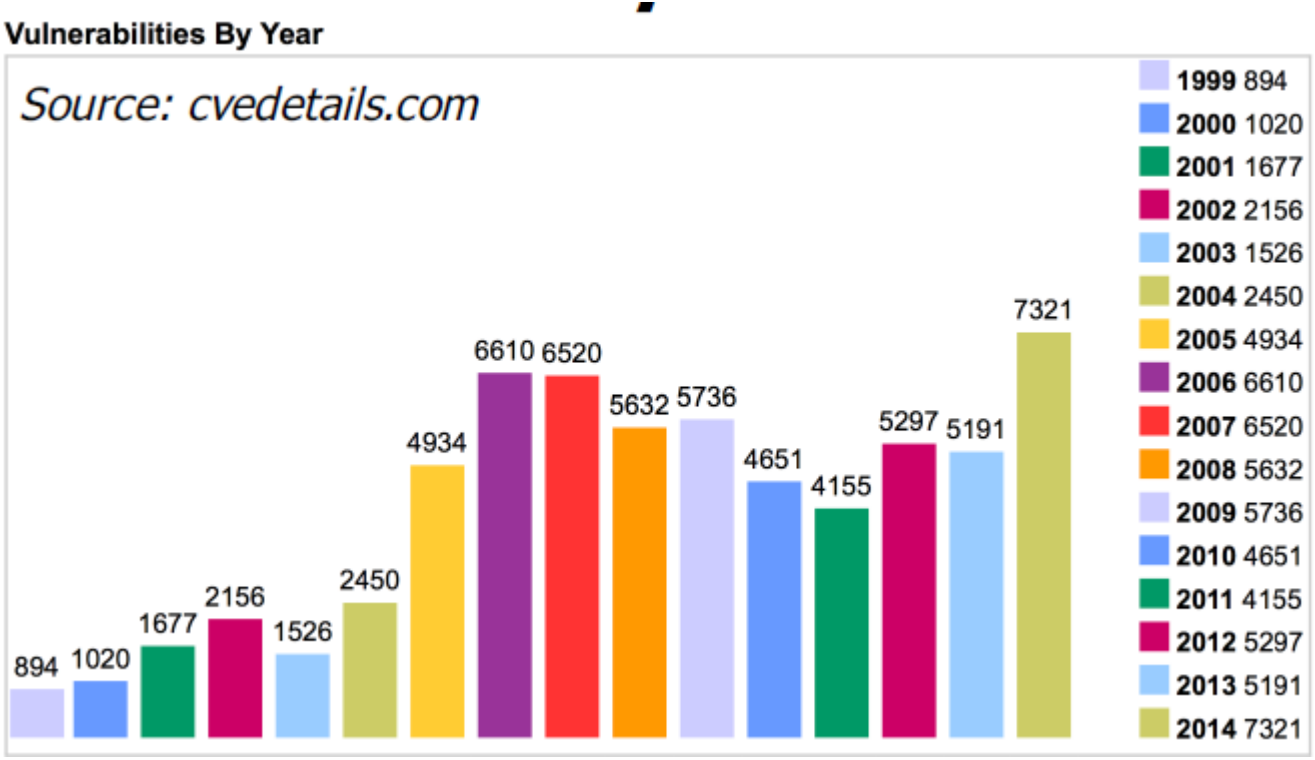
¿Cómo defendemos este pequeño detalle?



Defendiendo del exploit

- Indique de manera explícita el largo de la entrada con respecto al tamaño del buffer.
- Evitar llamadas que no revisan el largo de los datos.
  - Sprintf(buf, …), scanf(“%s”, buf)
- Mejor.
  - Snprintf(buf, buflen,…)
  - Scanf (“%256s”,buf)

Vulnerabilidad



Resumen

En resumen

- Nunca se confíen…
- En caso de que suceda algo, reportar de manera responsable y realizar el cambio.
- Piensen como atacantes.

Referencias

- Awad, A., & Karp, B. (2019, October). Execution integrity without implicit trust of system software. In *Proceedings of the 4th Workshop on System Software for Trusted Execution* (pp. 1-6).

¿Preguntas?

Realizado por:
Jason Leitón Jiménez.
Tecnológico de Costa Rica
Ingeniería en Computadores