Laboratorio 2

Bypass del ASLR: Brute-Force y trial-and-test

Máster en Ciberseguridad y Ciberinteligencia

Explotación de Binarios

Hector Marco

Universitat Politècnica de València

2024-25

Curso 2024-25

Introducción y objetivos

En Lab1 se demostró cómo ejecutar un exploit **ret2libc** con **NX** activo, basándose en direcciones fijas (`system`, `/bin/sh`). Sin embargo, si en el sistema está habilitada la **aleatorización** (**ASLR**), dichas direcciones cambian en cada ejecución, rompiendo el exploit.

En este Lab2 se abordan:

- Cómo **ASLR** afecta a un exploit ret2libc que funcionaba perfectamente con direcciones fijas.
- Ver por qué a veces, en entornos de **baja entropía** (p.ej. 8 bits, 256 posibilidades), es factible intentar repetidamente (trial-and-test) hasta "acertar" la base de `libc`.
- Discusión de un **servidor forking** que `fork()` por cada conexión, heredando la misma disposición de memoria en ciertos kernels (o con las mismas variables de entorno), y por qué en ese caso la repetición es más sencilla.

Requisitos y recordatorio

- Se asume que el exploit de Lab1 ("Parte 3: Bypass NX con ret2libc") funciona en 32 bits, **sin** ASLR o con direcciones fijas (ASLR=0 o binario sin PIE).
- Se recomienda repasar la teoría de ASLR (Sesión 2) pare entender mejor la entropía, configuraciones del kernel (`/proc/sys/kernel/randomize_va_space`), etc.

Parte 1: Caso ret2libc con ASLR activado

1.1. Confirmar que ret2libc funciona sin ASLR

1. **Desactivar** temporalmente ASLR:

   ```
   echo 0 > /proc/sys/kernel/randomize_va_space
   ```

2. Modifica el exploit del Lab1 para que ejecute el comando `ls` y verifica que sin tener el ASLR activado funciona.

1.2. Activar ASLR y observar el fallo

1. Reactiva ASLR:

   ```
   echo 2 > /proc/sys/kernel/randomize_va_space
   ```

2. Si estamos en un sistema de 64 bits, comprobar que tenemos 8-bits para mmap:

   ```
   cat /proc/sys/vm/mmap_rnd_compat_bits
   8
   ```

   Si obtenemos un número mayor a 8, lo configuramos para tenga solo 8:

   ```
   echo 8 >/proc/sys/vm/mmap_rnd_compat_bits
   ```

3. Ejecutar el **exploit** (misma dirección `system` y `ls`).
4. Observa que ahora obtendrás `Segmentation fault`.
5. Obtén una dirección de system y de la cadena `ls` de una ejecución con ASLR activado usando GDB. Para ello carga el programa en GDB pero habilitando el ASLR:

   ```
   (gdb) set disable-randomization off
   ```

6. Pon un breakpoint en main(), ejecuta el programa y cuando se pare en main(), muestra la dirección de `system()` desde el GDB:

   ```
   (gdb) print &system
   $10 = (<text variable, no debug info>*) 0xf7d178c0 <system>

   (gdb) find 0xf7d7c000, 0xf7f9a000, "ls"
   0xf7d931ae
   ```

7. Ejecuta el programa desde GDB varias veces y observa confirma que la dirección de `system()` y de donde está la cadena `ls` cambian.
8. Anota las direcciones de la última ejecución, ya que será utilizada para los siguientes ejercicios.

Conclusión: la **base de libc** está cambiando cada vez. El exploit original no se ajusta a esas direcciones dinámicas.

Parte 2: Trial-and-Test en 32 bits

En muchos sistemas de 32 bits, la entropía de ASLR puede ser baja (p.ej. **8 bits** para la base de `libc`), es decir, hay 256 posiciones posibles. En cada **nueva ejecución** de la aplicación, se vuelve a randomizar la dirección base donde están las librerías.

Un posible ataque es ejecutar el exploit muchas veces hasta que coincidamos con las direcciones obtenidas en una de las ejecuciones de GDB. Aunque esto no garantiza al 100 % el éxito, en la práctica funciona bien en sistemas de baja entropía.

2.1. Script de repetición (trial-and-test)

1. Crea un script `try.sh` que, en un bucle el exploit y pare cuando tenga éxito. La idea es: **cada vez** que se lanza el binario, se genera una nueva dirección base. El exploit está "fijo", esperando acertar.

```
#!/bin/bash

N=0
while true; do
   N=$((N+1))
   # Lanza exploit
   python3 exploit.py| ./vuln1
   if [ $? -eq 0 ]; then
      echo "Success on try #$N"
```

```
10        break
11    fi
12    echo "Failed on try #$N"
13 done
```

2. Ejecuta el script varias veces y observa el comportamiento. Si se queda "colgado" mata el proceso del bash script y vuelve a intentarlo. Comenta si todas las veces que lo lanzas se ejecuta el comando `ls` o no e intenta averiguar y explicar los comportamientos observados.

Parte 3: Escenario forking server

Haz un pequeño servidor que escuche en `localhost` y puerto `9999` que tenga una función llamada `vulnerable()` que tenga un desbordamiento de buffer.

- La aplicación debe llamar a `fork()` antes de llamar a vulnerable.
- El padre debe esperar a que el hijo termine (correctamente o no).
- Se debe compilar con 32-bits (`-m32`).
- Se debe ejecutar con el ASLR habilitado.
- Se debe compilar con NX.
- Se debe compilar sin el SSP.

Realiza ahora un exploit que ejecute un comando. El ataque debe:

- Implementar la estrategia de brute-force attack.
- Ejecutar algún comando.

Proporciona los detalles tanto del servidor como del exploit, así como los pasos realizados en la explotación.

Para entregar

1. **Informe (PDF)** con:
   - Resultados y pasos de todas las partes.
   - Código del servidor y de los scripts, así como los comandos de compilación y cualquier otro que sea necesario para reproducir los resultados.

Fin de Lab2. ¡Suerte!