Descubrimiento

Hacking Ético

©Ismael Ripoll &
Hector Marco

Universidad Politècnica de València

March 3, 2021

Índice

Qué vamos a trabajar

- ¿Cómo se encuentran los fallos?
- Es el trabajo de un pentester (Penetration tester) o un *vulnerability assesement*
- Veremos varias técnicas de búsqueda de “anomalías”.
- Es una “actitud”, más que una técnica.
- Muchas veces se encuentra de manera inesperada.

Pentester vs. vulnerability assesment

- **Un pentester** utiliza una base de datos de vulnerabilidades más su experiencia en los tipo de fallos que suelen cometer el tipo de sistema (empresa, equipo, red, etc.) objetivo para encontrar los fallos.
  Normalmente, los fallos encontrados son vulnerabilidades que **ya son conocidas** pero que no han sido parcheadas o son fallos relativamente genéricos.
  Lo suele hacer una empresa contra su propia infraestructura informática para conocer las debilidades de la misma.
- **Un vulnerability assesment** se centra en **nuevas** vulnerabilidades, aunque no ignora si se encontraran vulnerabilidades ya conocidas.
  Lo suele solicitar o contratar con el fabricante y se realiza en uno de sus productos, normalmente, antes de sacarlo al mercado para “adelantarse” a los potenciales atacantes.

Escaneado de red

- En un pentesting debemos identificar todos los equipos que forman el objetivo o que nos pueden ayudar a conseguir el objetivo.
- En un pentesting real, definir el scope y el objetivo no es trivial y es muy importante. Tiene un gran impacto en el tipo de pentesting y presupuesto del mismo, así como el equipo de personas que formar en el equipo.
- Para ello, necesitamos conocer la topología de la red y los equipos conectados.
- Hay que diferenciar entre:

  Escaneado de red: Conocer qué equipos hay en la red y qué servicios ofrecen.

  Escaneado de vulnerabilidades: Determinar qué programas o servicios pueden ser vulnerables.
- La mejor herramienta es para escanear redes **nmap**.

nmap (I)

`nmap` puede utilizar raw sockets con algunas opciones. Para ello necesita permisos de administrador. Entre otras muchas cosas, con nmap se puede:

- Escanear un host concreto:

```
# nmap localhost
Starting Nmap 7.01 ( https://nmap.org ) at 2018-05-29 16:40 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000060s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
631/tcp open ipp

Nmap done: 1 IP address (1 host up) scanned in 1.64 seconds
```

- Escanear rangos de IPs

```
# nmap 158.42.52.215/31
```

nmap (II)

- Escaneado “agresivo” para identificar los servicios y el tipo de sistema operativo:

```
# nmap -A localhost
Starting Nmap 7.01 ( https://nmap.org ) at 2018-05-29 16:34 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000070s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE VERSION
631/tcp open ipp   CUPS 2.1
| http-methods:
|_  Potentially risky methods: PUT
| http-robots.txt: 1 disallowed entry
|_/
|_http-server-header: CUPS/2.1 IPP/2.1
|_http-title: Inicio - CUPS 2.1.3
Device type: general purpose
Running: Linux 3.X|4.X
....
```

nmap (III)

- Listar los nombres DNS de rangos de IPs.

```
# host www.upv.es
www.upv.es is an alias for ias.cc.upv.es.
ias.cc.upv.es has address 158.42.4.23
ias.cc.upv.es mail is handled by 40 mx4.cc.upv.es.
ias.cc.upv.es mail is handled by 7 mxv.cc.upv.es.
ias.cc.upv.es mail is handled by 10 mx2.cc.upv.es.
ias.cc.upv.es mail is handled by 50 vega.cc.upv.es.
# nmap -sL 158.42.4.23/30

Starting Nmap 7.12 ( https://nmap.org ) at 2018-05-29 16:49 CEST
Nmap scan report for cali.cc.upv.es (158.42.4.20)
Nmap scan report for svndes.cc.upv.es (158.42.4.21)
Nmap scan report for www2.cc.upv.es (158.42.4.22)
Nmap scan report for ias.cc.upv.es (158.42.4.23)
Nmap done: 4 IP addresses (0 hosts up) scanned in 0.00 seconds
```

Podemos ver que conocemos los nombres de los ordenadores “vecinos” de www.upv.es.

nmap (IV)

- Tiene infinidad de opciones para ajustar la forma de realizar el escaneado.
- A raíz de las protecciones de los firewalls, nmap implementa múltiples estrategias para identificar si un host o un puerto están operativos. Por ejemplo el flag `-Pn` indica que realice el escaneado incluso si el host no responde al ping.
- Otra característica interesante es que ofrece varios formatos de salida que nos permitirán procesar la salida fácilmente: `-oG` (*grepable*), `-oX` (XML) y `-oS` (divertida).

Fuzzing (I)

Una vez conocemos el entorno el siguiente paso es buscar fallos de forma remota.

Fuzzing [Wikipedia]

Fuzzing es una técnica de pruebas de software, a menudo automatizado o semiautomatizado, que implica proporcionar datos inválidos, inesperados o aleatorios a las entradas de un programa de ordenador.

- Los programadores suelen diseñar programas que son utilizados por humanos o por otros programas. Por lo que solo se contemplan y validan los tipos de fallos que normalmente comenten los humanos y se asume que las máquinas no se equivocan.

Fuzzing (II)

- Los fuzzers se suelen comportar como usuarios “atolondrados” muy rápidos, por lo que pueden probar muchas combinaciones imprevistas por el programador.
- El principal problema de los fuzzers automáticos es poder determinar si un resultado es correcto (el target ha detectado que la entrada es inválida) o se trata de una fallo no gestionado por el target y por tanto es una potencial vulnerabilidad.
- Muchos fuzzers se desarrollan como herramientas de testeo de software.
- Cada tipo de target (aplicación, servidor, etc.) necesita de un fuzzer especializado. En muchas ocasiones se diseñan ad hoc.
- Por ejemplo: `crashme` es un fuzzer para probar la robustez del kernel de Linux realizando llamadas al sistema con parámetros absurdos.

Fuzzing (III)

- Si lo que se quiere probar tiene controles de integridad, por ejemplo los paquetes TCP/IP (con el CRC), entonces es necesario que el fuzzer construya paquetes válidos.
- Algunos fuzzers tienen un patrón o plantilla base sobre la que hacen modificaciones aleatorias y luego ajustes para evitar las inconsistencias que trivialmente detectará el target.
- Si el target mantiene un estado, entonces puede ser que alguna de las pruebas anteriores interfiera con resultados posteriores. Lo que puede dificultar la "reproducibilidad" del fallo.
- Cuando el fuzzing es manual recibe el nombre de "*mokey testing*".

Técnicas

Mutación: A partir de una entrada válida se realizan modificación sobre ella de forma ciega. No considera la estructura de los datos.
Son lentos pero pueden encontrar fallos difíciles.

Generación: A partir de una especificación del modelo de los datos de entrada, se construyen entradas válidas. Si considera la entrada de los datos.
Con este tipo de pueden encontrar SQL injections con relativa facilidad.

Repetición: Se trata de expandir los valores de entrada (longitud de las cadenas) para encontrar buffer overflows.

Ejemplos (I)

AFL: American Fuzzy Lop es un fuzzer gratuito muy utilizado.

sqlmap: Interesante programa (en python)

ZAP: Utilidad desarrollada por OWASP. Es mucho más que un fuzzer, es un proxy web.

crashme: Antes comentado.

fuzz: Simple fuzzer para entrada estandar y argumentos.

Pero si nosotros podemos utilizar un fuzzer (off-the-shelf) entonces, los desarrolladores también, al igual que otros atacantes.
Por tanto: los fuzzers públicos no soy muy útiles para descubrir vulnerabilidades interesantes.

Troleando a sqlmap (I)

La mejor forma de ver como funciona un fuzzer es mirando qué y cómo hace las pruebas. Vamos a construir un escenario para verlo:

1. Construimos un fichero http válido:

```
$ wget -S www.upv.es
-2018-05-31 17:33:11-- http://www.upv.es/
Resolviendo www.upv.es (www.upv.es)... 158.42.4.23
Conectando con www.upv.es (www.upv.es)[158.42.4.23]:80... conectado.
Petición HTTP enviada, esperando respuesta...
  HTTP/1.1 200 OK
  Date: Thu, 31 May 2018 15:33:14 GMT
  Server: Apache
  AMFplus-Ver: 1.4.0.0
  Accept-Ranges: bytes
  X-UA-Compatible: IE=EmulateIE7, IE=11
  Connection: close
  Transfer-Encoding: chunked
  Content-Type: text/html; charset=ISO-8859-1
  Content-Language: es
Longitud: no especificado [text/html]
Grabando a: "index.html".1
```

Troleando a sqlmap (II)

Le quitamos la cabecera de “Transfer-Encoding: chunked”.

2. Levantamos un servidor (escuchador) tonto:

```
$ for i in {0..10000} ; do nc -l 9090 < http; done | grep GET
```

3. En otra consola lanzamos el ataque contra nosotros mismos:

```
$ sqlmap --level=4 --batch -u "http://localhost:9090/id=1"
```

4. En la consola del servidor (proceso nc) veremos:

```
GET /id=1 HTTP/1.1
GET /id=1 HTTP/1.1
GET /id=7147 HTTP/1.1
GET /id=1.%2C%2C%29%28%27%29%27%29%29 HTTP/1.1
GET /id=1%27uClSpI%3C%27%22%3EaZicsx HTTP/1.1
GET /id=1%29%20AND%206281%3D1036 HTTP/1.1
GET /id=1%27%20AND%207248%3D6810 HTTP/1.1
GET /id=1%27%20AND%209449%3D9449 HTTP/1.1
GET /id=1%29%20AND%205639%3D5804%20AND%20%288796%3D8796 HTTP/1.1
....
```

Código fuente (I)

- Otra forma de encontrar vulnerabilidades es directamente estudiando el código fuente.
- Esto se puede hacer en proyectos open source, o si se cuenta con un buen des-compilador.
- Se puede hacer una búsqueda manual o semiautomática.
- ¿Qué se busca? Pues todos los tipos de fallos enumerados en la lista de CWE!
  1. Variables no inicializadas.
  2. Desbordamientos de buffer.
  3. Condiciones de carrera.
  4. Secretos hardcodeados.
  5. Gestión de memoria dinámica incorrecta: use-after-free, double-free.
  6. No comprobar los valores de retorno.
  7. Falta de sanitización de los datos de usuario.

Código fuente (II)

8. Uso de funciones vulnerables.
9. Mal uso de criptografía.
10. Ejecución con excesivos permisos.
11. etc...

- Es un buen ejercicio estudiar el código de proyectos de software libre.
- La mayoría de ellos con excelentes ejemplos de buena programación... pero no todos, claro.

Caso real: GRUB28

- Decidimos estudiar cuál era el código que gestiona **la primera barrera que presenta Linux** en un sistema altamente protegido.
- Lo primero que nos encontramos es con el cargador de sistemas operativos GRUB2.
- Te bajas los fuentes y a inspeccionar código (`grep`, `find`, ...)

```
$ mkdir analisis
$ cd analisis
$ apt source grub2
$ ls -l
drwxrwxr-x 15 test test 4096 jun 7 16:26 grub2-2.02~beta2
-rw-r--r-- 1 test test 1060680 jun 7 2017 grub2_2.02~beta2-36ubuntu11.3.
     debian.tar.xz
-rw-r--r-- 1 test test 6354 jun 7 2017 grub2_2.02~beta2-36ubuntu11.3.dsc
-rw-r--r-- 1 test test 5798740 ene 17 2014 grub2_2.02~beta2.orig.tar.xz
$ grep -r _password_
...ib/crypto.c:grub_password_get (char buf[], unsigned buf_size)
...ests/lib/functional_test.c: grub_dl_load ("legacy_password_test");
...ests/legacy_password_test.c:legacy_password_test (void)
...ests/legacy_password_test.c:GRUB_FUNCTIONAL_TEST (legacy_password...
```

```
static int grub_username_get (char buf[], unsigned buf_size) {
  unsigned cur_len = 0; int key;
  while (1) {
      key = grub_getkey ();
      if (key == '\n' || key == '\r') break;
      if (key == '\e') {
          cur_len = 0;
          break;
       }
      if (key == '\b') {     // Si pulsas Retroceso entonces
          cur_len--;         //   decrementa cur_len (si o si)
          grub_printf ("\b"); // Esto huele a integer overflow ;-)
          continue;
      }
      if (!grub_isprint (key)) continue;
      if (cur_len + 2 < buf_size) {
          buf[cur_len++] = key; // Off-by-two !!
          grub_printf ("%c", key);
      }
  }
  grub_memset( buf + cur_len, 0, buf_size - cur_len);
  grub_xputs ("\n");
  grub_refresh ();
  return (key != '\e');
}
```

1-days (I)

- Una vez se hace pública o se reporta una vulnerabilidad, esta es resuelta ¿No?
- Pues no siempre. Por ejemplo, existe un importante GAP entre los fallos conocidos en Android y los que nuestros teléfonos tienen resueltos.
- La ventana temporal de explotación puede ser desde cero días a años. Microsoft no parcheó en varios años un grave fallo en la versión en castellano del IIS que permitía hacer un *pathtraversal* con ejecución de código. **Este fallo afectó a los servidores de la universidad... durante meses.**
- El ataque del "Wannacry" contra Telefonica de 2017 explotaba un fallo en windows XP(r) conocido que Microsoft(r) no había corregido por ya no estar soportado.