Detectando corrupción de memoria con Ptrace en aplicaciones reales

Gustavo Grieco

Universidad Nacional de Rosario - CIFASIS-Conicet

XII Jornadas de Ciencias de la Computación

Esquema general

1. Motivación: cuando la memoria se corrompe
   - Definición
   - Ejemplo
2. Ocean: el examinandor de trazas
   - Sistemas Operativos
   - Ptrace
   - Ltrace
3. Ejercicios: el principio del fin

Plan

Una definición práctica

- Lectura/escritura de buffers fuera sus límites
- Uso de punteros inválidos (por ejemplo, nulos)
- Uso de memoria no inicializada
- Uso de objetos en memoria liberada (por ejemplo, use-after-free)
- Liberación de memoria ilegal (por ejemplo, double-free)

Error:

Terminación anormal de un programa

Vulnerabilidades

Ejecución arbitraria de código

Seguridad en programas 101

```c
int main (int argc, char *argv[]) {
  char user[128];
  char cmd[128];
  char buffer[1024];

  strcpy(cmd, "./show_users");
  strcpy(user, argv[1]);

  FILE* stream = popen(cmd, "r");
  fread (buffer, 1, 1023, stream);
  buffer[1023] = NULL;

  if (strstr(buffer, user)) {
    printf("user \"%s\" found in system\n", user);
  }

  pclose(stream);
  return 0;
}
```

Normalmente:

prog "root" → user "root" found in system

Desafortunadamente:

prog "aaa..." → sh: 1: aaa...: not found

Seguridad en programas 101

```
int main (int argc, char *argv[]) {
  char user[128];
  char cmd[128];
  char buffer[1024];

  strcpy(cmd, "./show_users");
  strcpy(user, argv[1]);

  FILE* stream = popen(cmd, "r");
  fread (buffer, 1, 1023, stream);
  buffer[1023] = NULL;

  if (strstr(buffer, user)) {
    printf("user \"%s\" found in system\n", user);
  }

  pclose(stream);
  return 0;
}
```

Normalmente:

prog "root" → user "root" found in system

Desafortunadamente:

prog "aaa..." → sh: 1: aaa...: not found

Seguridad en programas 101

```
int main (int argc, char *argv[]) {
  char user[128];
  char cmd[128];
  char buffer[1024];

  strcpy(cmd, "./show_users");
  strcpy(user, argv[1]);

  FILE* stream = popen(cmd, "r");
  fread (buffer, 1, 1023, stream);
  buffer[1023] = NULL;

  if (strstr(buffer, user)) {
    printf("user \"%s\" found in system\n", user);
  }

  pclose(stream);
  return 0;
}
```

Normalmente:

prog "root" → user "root" found in system

Desafortunadamente:

prog "aaa..." → sh: 1: aaa...: not found

Inspeccionando trazas

- Secuencias de eventos:
  - Llamadas a funciones con todos sus argumentos (strcpy, fread, ..)
  - Estado final (exit, crash, abort or timeout)

- prog "root"
  1. strcpy
  2. strcpy
  3. popen
  4. fread
  5. strstr
  6. printf
  7. pclose
  8. **exit**

- prog "roAAAAot"
  1. strcpy
  2. strcpy
  3. popen
  4. fread
  5. strstr
  6. pclose
  7. **exit**

- prog
  1. strcpy
  2. strcpy..
  3. **crash**

Plan

Introducimos Ocean

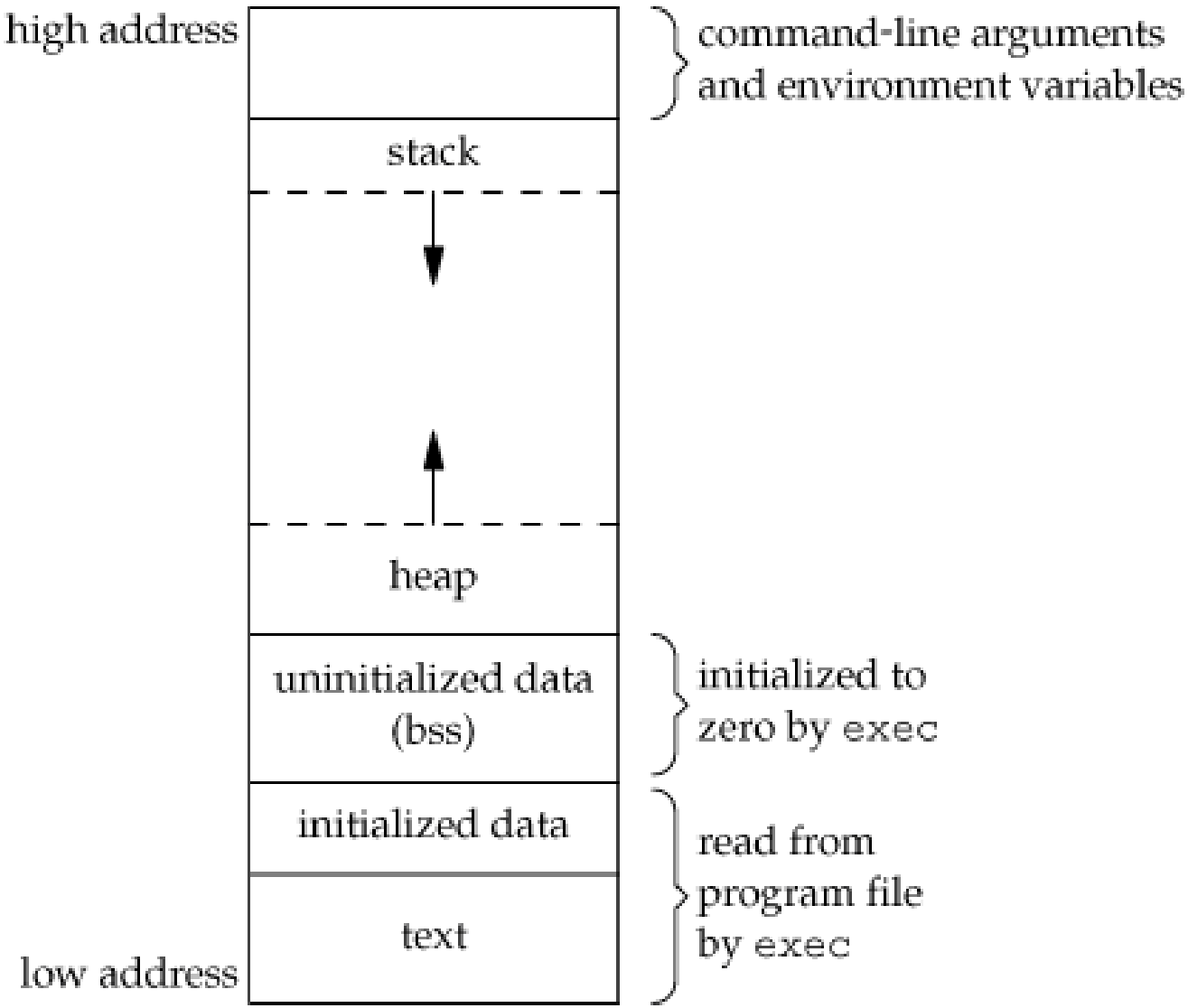
- Herramienta de **análisis dinámico** de procesos.
- Analiza **trazas de eventos** obtenidas de **ejecuciones**.
- Similar a **ltrace**, pero más poderoso y extensible.
- Desarrollada en Python, usando python-ptrace.
- Software Libre (GPL3), sin dependencias propietarias.
- Creada para mi doctorado, como un subproducto útil (?)

`https://github.com/neuromancer/ocean`

Cuestiones básicas

- Ejecutable: Archivo con un programa en forma de instrucciones máquina listas para ser ejecutadas.
- Proceso: Una instancia de un programa ejecutable en en memoria, junto con su estado.
- Espacio de memoria: Conjunto de páginas de memoria reservadas para un determinado proceso.

La memoria de un proceso en detalle



Compartiendo código entre varios procesos

- Librerías: Archivo con instrucciones en lenguaje máquina compartido por varios ejecutables.
  - Estáticas: el código ejecutable se copia y pega en los distintos ejecutables, para luego cargarse varias veces.
  - Dinámicas: el código ejecutable se carga en memoria una vez y se reutiliza allí.

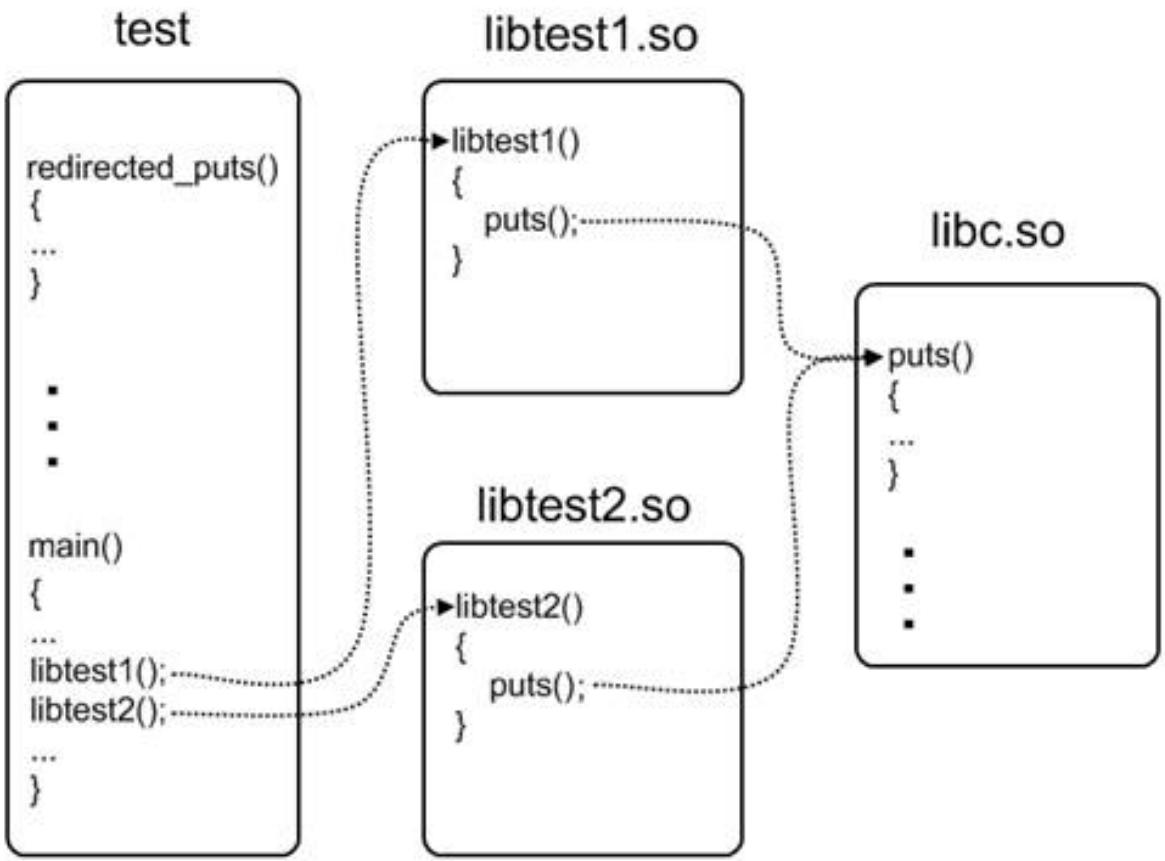
En GNU/Linux:

- Librerías estáticas: .a
- Librerías dinámicas: .so

En Windows:

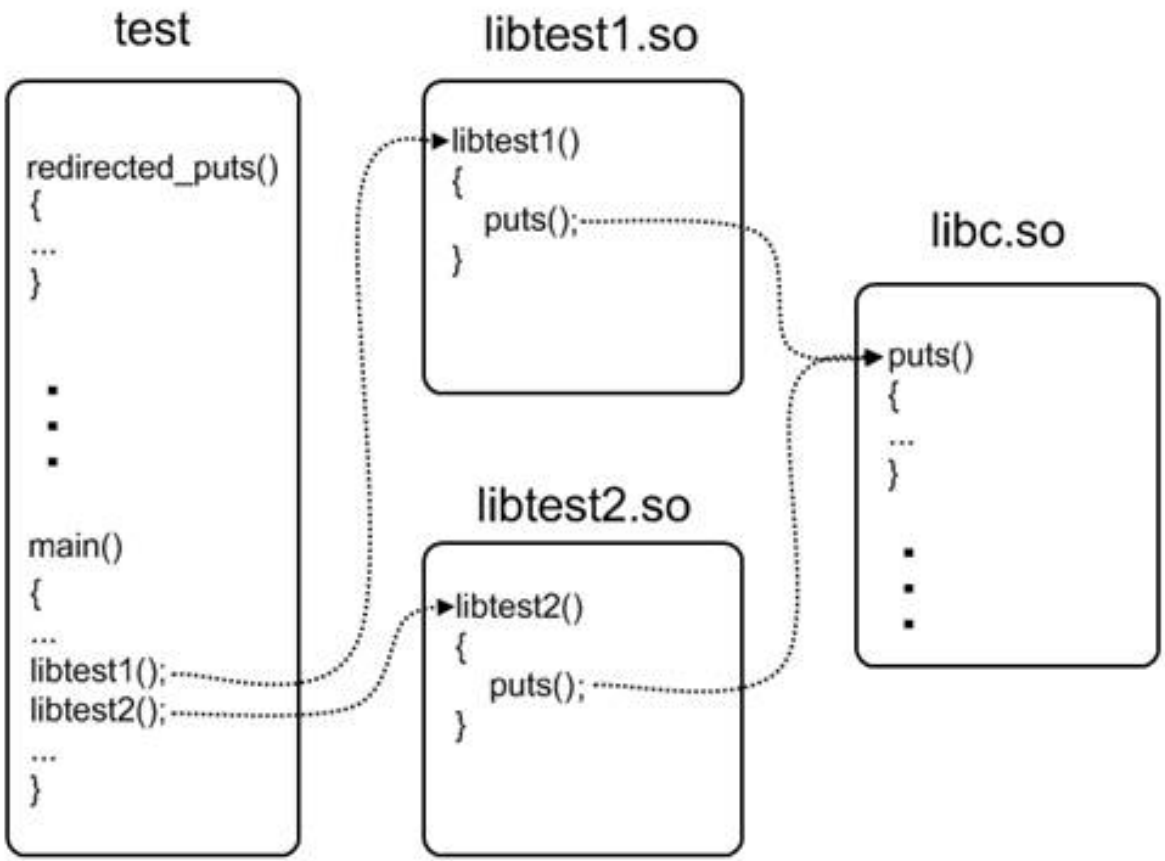
- Librerías estáticas: .lib
- Librerías dinámicas: .dll

Por ejemplo..



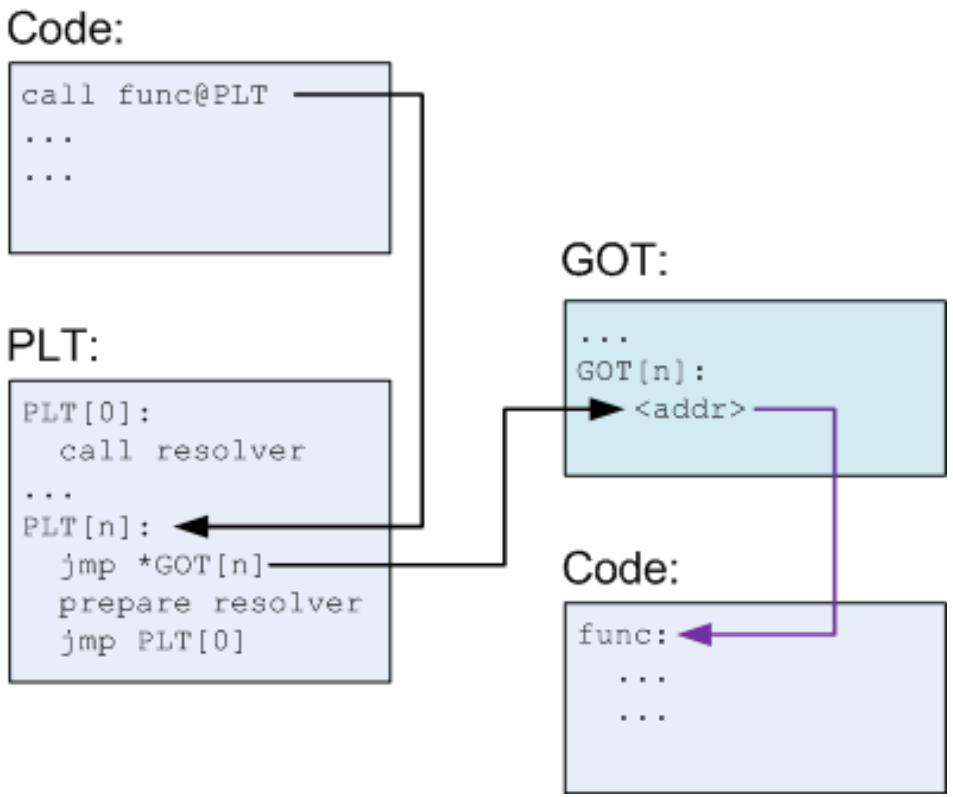
¿Qué son esas flechas?

Por ejemplo..



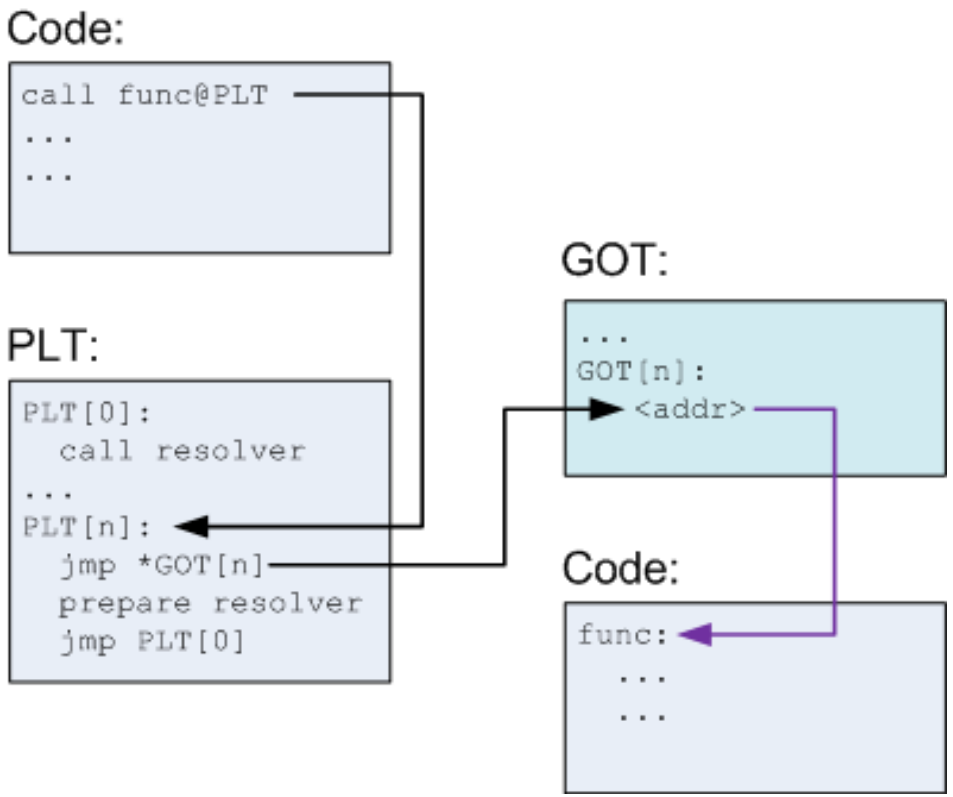
¿Qué son esas flechas?

Unos detalles más sobre llamadas a librerías dinámicas



Ahora veamos un poco de un mecanismo para depuración en Linux..

Unos detalles más sobre llamadas a librerías dinámicas



Ahora veamos un poco de un mecanismo para depuración en Linux..

Process Trace, a.k.a. **Ptrace**

Ptrace

- Definición: Llamada a sistema para observar y manipular un proceso.
- Ventajas:
  - Interfaz minimalista y elegante.
  - Mínima pérdida de velocidad.
  - No requiere privilegios de root.
- Desventajas:
  - Interfaz rígida.
  - Sólo *NIX (Linux, BSD, SunOS, ..)
  - Debe ser limitado para procesos con suid.

PTrace in a nutshell

```
#include <sys/ptrace.h>

long ptrace (enum __ptrace_request request,
             pid_t pid,
             void *addr,
             void *data);
```

Vayamos por partes:

- __ptrace_request request: La petición de ptrace que indica la operación a realizar
- pid_t pid: El identificador de procesos donde realizar la operación
- void *addr: La dirección de memoria donde leer o escribir para la operación.
- void *data: La dirección de memoria donde leer o escribir datos del proceso analizado.

Peticiones (I)

- Control sobre procesos:
  - PTRACE_TRACEME, PTRACE_ATTACH, PTRACE_DEATTACH
- Lectura de memoria (por bytes, words y double-words):
  - PTRACE_PEEKTEXT, PTRACE_PEEKDATA
- Escritura de memoria (por bytes, words y double-words):
  - PTRACE_POKETEXT, PTRACE_POKEDATA

Peticiones (II)

- Si un proceso está detenido..
  - PTRACE_CONT para continuar ejecutando.
  - PTRACE_SYSCALL para continuar ejecutando hasta la siguiente llamada a sistema.
  - PTRACE_SINGLESTEP para continuar ejecutando hasta la siguiente instrucción.

¡Python al rescate!

Python-Ptrace

- Lectura y escritura de memoria en procesos.
- Interfaces de alto nivel (PtraceDebugger y PtraceProcess).
- Breakpoints.
- Señales.
- Desensamblado de instrucciones (opcional).
- Ejemplos completos de uso (strace.py y gdb.py)

- Soporte multiplataforma (Linux, FreeBSD, OpenBSD)
- Soport multiarquitectura (x86, x86_64 (Linux), PPC (Linux), ARM (Linux EAPI))

Breakpoints en Python-Ptrace

```
class Breakpoint(object):
    """ Software breakpoint. """
    def __init__(self, process, address, size=None):

        ...

        # Store instruction bytes
        self.old_bytes = process.readBytes(address, size)

        if CPU_POWERPC:
            # Replace instruction with "TRAP"
            new_bytes = word2bytes(0x0cc00000)
        else:
            # Replace instruction with "INT 3"
            new_bytes = b("\xCC") * size

        process.writeBytes(address, new_bytes)
        self._installed = True
```

Library Trace, a.k.a. **Ltrace**

Ltrace con un ejemplo..

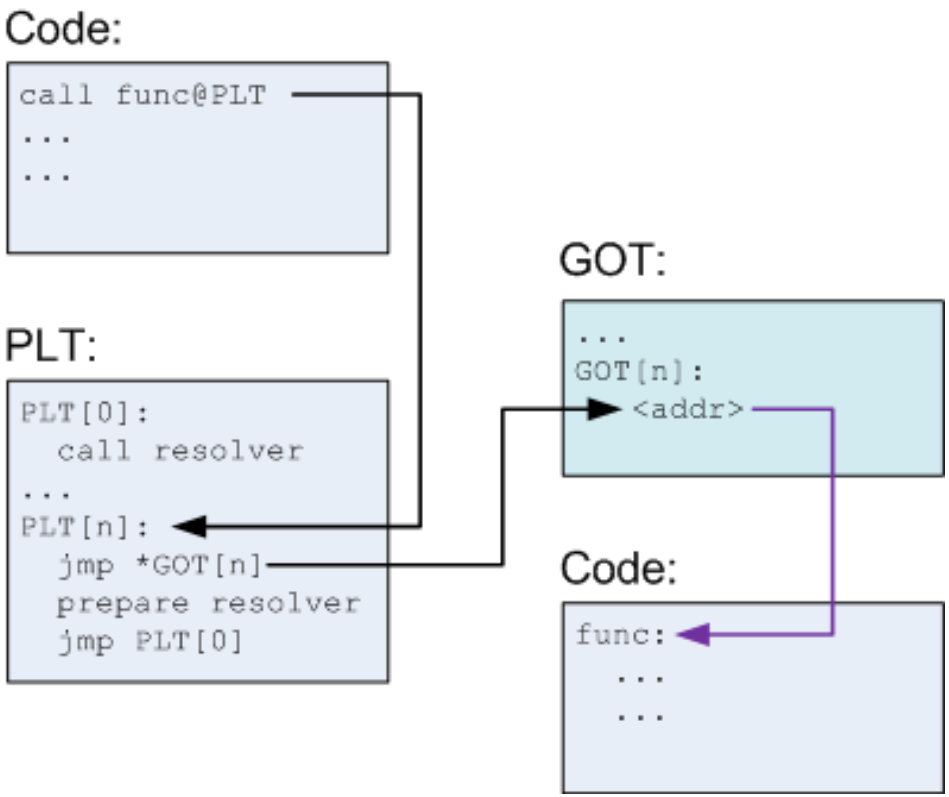
Ltrace es una utilidad de depuración para mostrar los llamados de las librerías dinámicas que hace un ejecutable en Linux. Por ejemplo:

```
$ ltrace echo "abc"
getenv("POSIXLY_CORRECT") = nil
strrchr("echo", '/') = nil
setlocale(LC_ALL, "") = "LC_CTYPE=en_US.UTF-8;LC_NUMERIC="...
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils") = "coreutils"
__cxa_atexit(0x80497f0, 0, 0, 0xbffff3b4, 0xbffff308) = 0
strcmp("abc", "--help") = 52
strcmp("abc", "--version") = 52
fputs_unlocked(0xbffff563, 0xb7fb34e0, 0xb7fb3ce0, 0x8048b2c, 0xb7ff0590) = 1
__overflow(0xb7fb34e0, 10, 0xb7fb3ce0, 0x8048b2c, 0xb7ff0590abc = 10
exit(0 <unfinished ...>
```

Prototipos de funciones conocidas

```
; string.h
void bcopy(addr,addr,ulong);
void bzero(addr,ulong);
string basename(string);
string index(string,char);
addr memchr(string,char,ulong);
addr memcpy(addr,string3,ulong);
addr memset(addr,char,long);
string rindex(string,char);
...
```

¿Cómo funciona ltrace?



¿Donde ponemos el breakpoint?

- call func@PLT
- jmp *GOT[n]
- func

Un poco más de detalle sobre Ocean

- Especificar distintos tipos de entradas para programas:
  - Argumentos
  - Archivos
  - Entrada estándar
- Detectar eventos en trazas mediante breakpoints (a la ltrace)
- Analizar direcciones de memoria dinámicamente:
  - Clasificar valores de los argumentos según su "tipo"
  - Detectar algunos tipos de corrupción de memoria.
- Mutar los casos originales para generar (con un poco de suerte) nuevas trazas.
- Ser extendido fácilmente (por estar hecho en Python!)

Especificando casos de pruebas en Ocean

"The Yasm Modular Assembler is a portable, retargetable assembler written under the "new" (2 or 3 clause) BSD license. ytasm is the TASM- compatible frontend."

Dentro del directorio ytasm-testcase:

path.txt	→	`/usr/bin/ytasm`
`crash/argv_1.symb`	→	`/DD[DAsAAA`
`crash/file___dev__stdin.symb`	→	`AAAAAAAAAAAAAAAAAAAAAAAA`

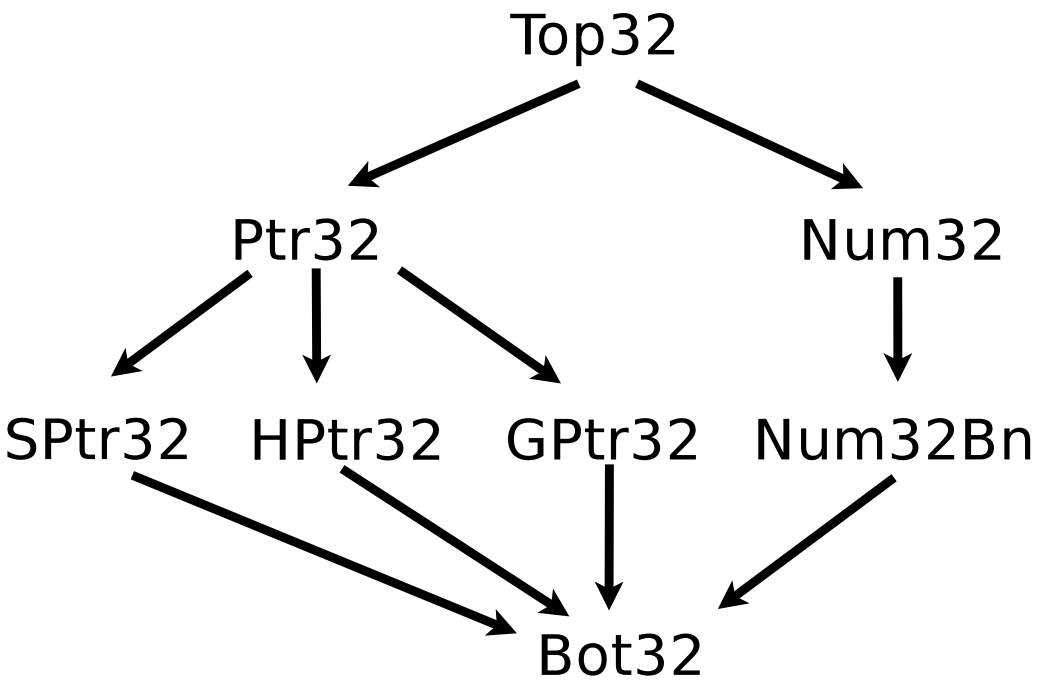
En la herramienta se ejecuta:

```
/usr/bin/ytasm /DD[DAsAAA < file___dev__stdin.symb
```

Analizando el crash de ytasm

```
$ ocean.py ytasm-report
WARNING:root:Terminate <PtraceProcess #4644>
/usr/bin/ytasm type=null textdomain:0=GxPtr32 malloc:0=Num32B8
malloc:0=Num32B8 malloc:0=Num32B8 malloc:0=Num32B8 ... malloc:0=Num32B8
malloc:0=Num32B8 strlen:0=GxPtr32 strlen:0=GxPtr32 strlen:0=GxPtr32
strlen:0=GxPtr32 strlen:0=GxPtr32 strlen:0=GxPtr32 malloc:0=Num32B8
SIGSEGV:addr=Top32 crashed:eip=GxPtr32
```

Sistemas de tipos para punteros y enteros

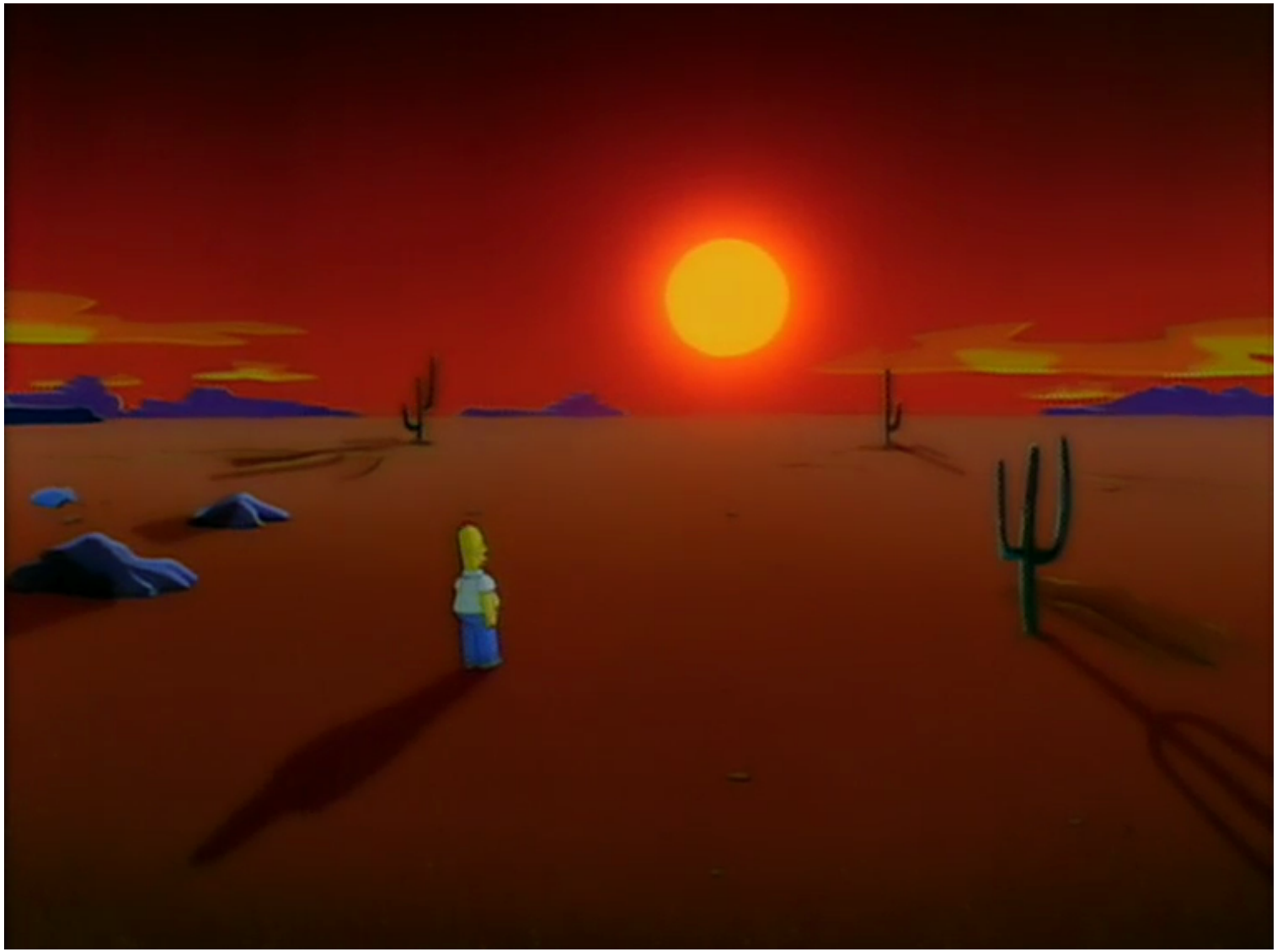


tipo	rango entero
Num32B0	$[0,2^0)$
Num32B1	$[1,2^1)$
Num32B2	$[2^1+1,2^2)$
...	...
Num32B32	$[2^{31}+1,2^{32})$

Detectando vulnerabilidades

```
def detect_vulnerability(preevents, event, process, mm):

    if isinstance(event, Call):

      ...

    elif isinstance(event, Abort):

      ...

    elif isinstance(event, Crash):

      if str(event.fp_type[0]) == "DPtr32" and str(event.eip_type[0])
        return Vulnerability("StackCorruption")

      for (typ,val) in event.bt:
        if str(typ) == "DPtr32":
          return Vulnerability("StackCorruption")

    elif isinstance(event, Signal):
      pass

    return None
```

Llego el momento.. de los ejercicios



Plan

Preparando el terreno..

1. **http://git.io/Y6NfNA** (para no tener que copiar la linea de abajo)
2. git clone https://gist.github.com/neuromancer/ee8c76064a0e44385736 ocean-workshop
3. cd ocean-workshop; ./start.sh
4. vagrant up
5. vagrant ssh
6. cd ocean ; git pull
7. showtime!

Ejercicios (aperitivos)

- Examine la clase TypePrinter y agregue la siguiente funcionalidad accesible con nuevos flags en el ejecutable de la herramienta:
  - La posibilidad de mostrar valores (enteros, punteros, etc) de los eventos en vez de sus tipos. Para esto defina una clase llamada ValuePrinter. La clase TypePrinter implementa un filtro para no mostrar trazas repetidas. ¿Es conveniente implementar la misma funcionalidad en ValuePrinter?
    - Archivos sugeridos para modificar: src/Printer.py, ocean.py
  - La posibilidad de mostrar la dirección donde se originó el Call. La sintaxis sugerida es: malloc:0@0xdeadbeef=Num32B8
    - Archivos sugeridos para modificar: src/Printer.py, src/Event.py, ocean.py
  - La posibilidad de ver el módulo que genera cada evento. La sintaxis sugerida es: malloc:0:/usr/lib/somelib.so=Num32B8
    - Archivos sugeridos para modificar: src/Printer.py, src/Event.py, ocean.py

Ejercicios (más aperitivos)

- Implemente en la clase ValuePrinter la posibilidad de mostrar strings de acuerdo a los tipos definidos en las especificaciones en C de las funciones. ¿Cómo deben tratarse los strings de longitud muy larga? Agregue una flag para controlar el tamaño máximo de los string a imprimir.
  - Archivos sugeridos para modificar: src/Printer.py, src/Events.py, ocean.py
- Implemente la detección de corrupción de memoria en funciones que ejecutan programas (popen, system, exec*), para esto primero defina una heurísticas más o menos efectiva para detectar llamadas a programas "sospechosas" y luego modifique la función detect_vulnerability para agregar un evento especial en caso de detectar este tipo de situaciones.
  - Archivos sugeridos para modificar: src/Vulnerabilities.py, src/Events.py
- Revise el código fuente del paquete fuente de ytasm y localice la corrupción de memoria que genera el crash. ¿Ha sido este error solucionado en la ultima versión del paquete?

Ejercicios (platos principales)

- Instrumente las funciones que escriben buffers en memoria (strcpy, strncpy, memcpy, memset, ...) para verificar que cada vez que se escribe se haga en dentro de un buffer y no se corrompa la memoria. Lógicamente, una definición que cubra el 100 % de estos errores es impráctica/imposible, pero es posible definir algunas heurísticas **seguras** para detectar las lecturas/escrituras en memoria que violan la seguridad de la memoria. Defina criterios para:
  - Heap
  - Stack
  - ¿Global?
- Pruebe las modificaciones hecha a Ocean con los casos provisto en la maquina virtual. Una lista de programas "interesantes" puede encontrar en el archivo "los_sospechosos_de_siempre.txt"