```
 _____    ___   ___  _____
|  _  |  /   | /   ||___  /
| |_| | / /| |/ /| |   / /
\____ |/ /_| / /_| |  / /
.___/ /\___  \___  | / /
\____/     |_/   |_//_/
```

-=[**HEAP SPLOIT'N**]=-

by epochfail

i totally stole
the title slide

08-SEP-15

also the rest of ‘em

id

- Respond to “epoch”, “ev”, “russian”
- UNSW graduate
- Non-security eng @ Google
- Plumber at https://9447.plumbing/

id

- Respond to “epoch”, “ev”, “russian”
- UNSW graduate
- Non-security eng @ Google
- Plumber at https://9447.plumbing/
- Sometimes cosplay as a pwny

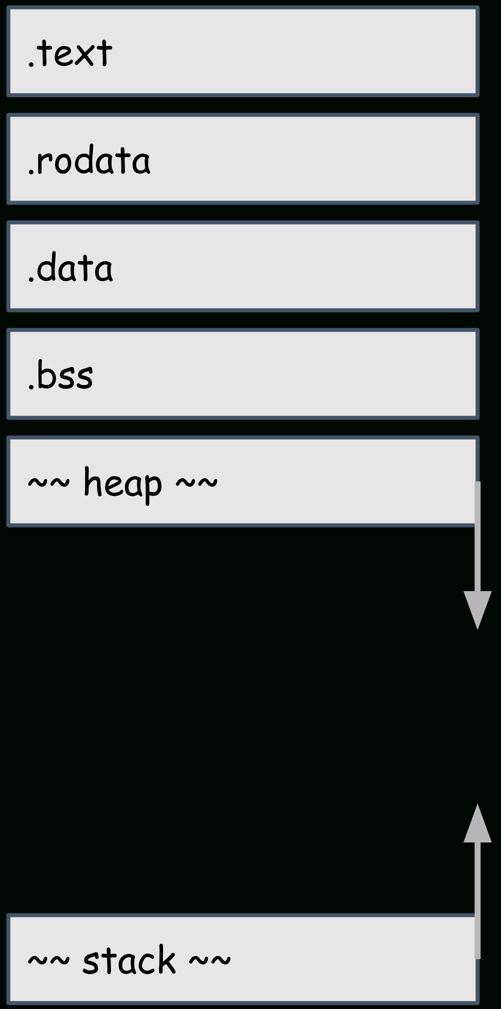
- Nothing I say is representative of my employer, etc, etc

# summary

- wat is heap
- types of bugs
- live haq demo
- glibc y u so hard

All I wanted to do is ~~make video~~
play wargames.
WTF is a malloc?
quickmeme.com

# address space



.text

.rodata

.data

.bss

~~ heap ~~

~~ stack ~~

```
$ cat /proc/`pidof saas-stripped`/maps
08048000-080ef000 r-xp ...... saas-stripped
080ef000-080f1000 rwxp ...... saas-stripped
080f1000-080f3000 rwxp
089a2000-089c4000 rwxp        [heap]
f778a000-f778b000 rwxp
f778b000-f778c000 r-xp        [vdso]
ffe09000-ffe2a000 rwxp        [stack]

^^ addresses ^^   ^^ perms    ^^ what’s inside
```

# basics

- OS provides memory in 4kb multiples (0x1000)
- Need a way to partition these regions
- Usually a linked list on top of those regions

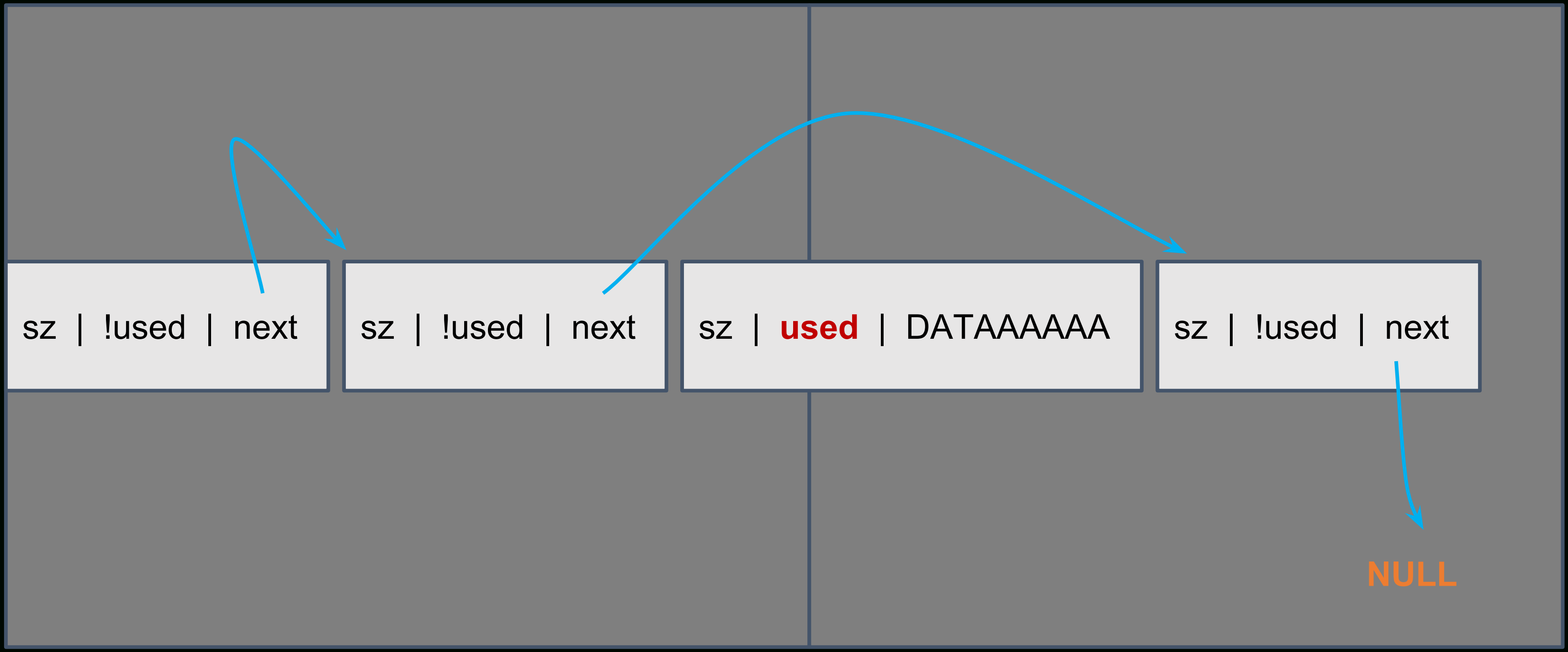
# basics

- void* malloc(size_t sz)   -- allocates sz bytes in heap
- void free(void* ptr)   -- returns malloced memory
- calloc(), realloc()   -- similar

- alloca()   -- allocates on stack (not heap)

# basics

Fictional heap (may not make sense)



# ezhp

- Plaid CTF 2014, 200 pts

- Note-keeping system
  - Add note
  - Read note
  - Modify note
  - Delete note

- Reversing is left as an exercise to the reader (or a later lecture)

-
  ```
  struct Node {
      int size;               // <-- metadata
      Node *next, *prev;
      char data[];            // <-- user data goes here
  }
  ```

# decompiled (not really)

```c
void deallocate(void *v) {
  if (!v) return;
  mm_header *curr = (mm_header *)((char *)v - sizeof(mm_header));
  mm_header *prev = curr->prev;
  mm_header *next = curr->next;

  // we don't bother coalescing.
  if (prev) prev->next = next;
  if (next) next->prev = prev;
  curr->next = base->next;

  if (base->next) base->next->prev = curr;
  base->next = curr;
  curr->sz &= ~1;
}
```

For allocate(), etc:
https://github.com/pwning/plaidctf2014/blob/master/pwnables/ezhp/ezhp.c

# unlink() sploit'n

- Old-school technique (ceyx/dzhkh were alive back then). Patched in 2004 or so

- Write-what-where primitive
  - Can write a value to an address of choice
  - Here, write pointer to data under our control

# unlink() sploit'n

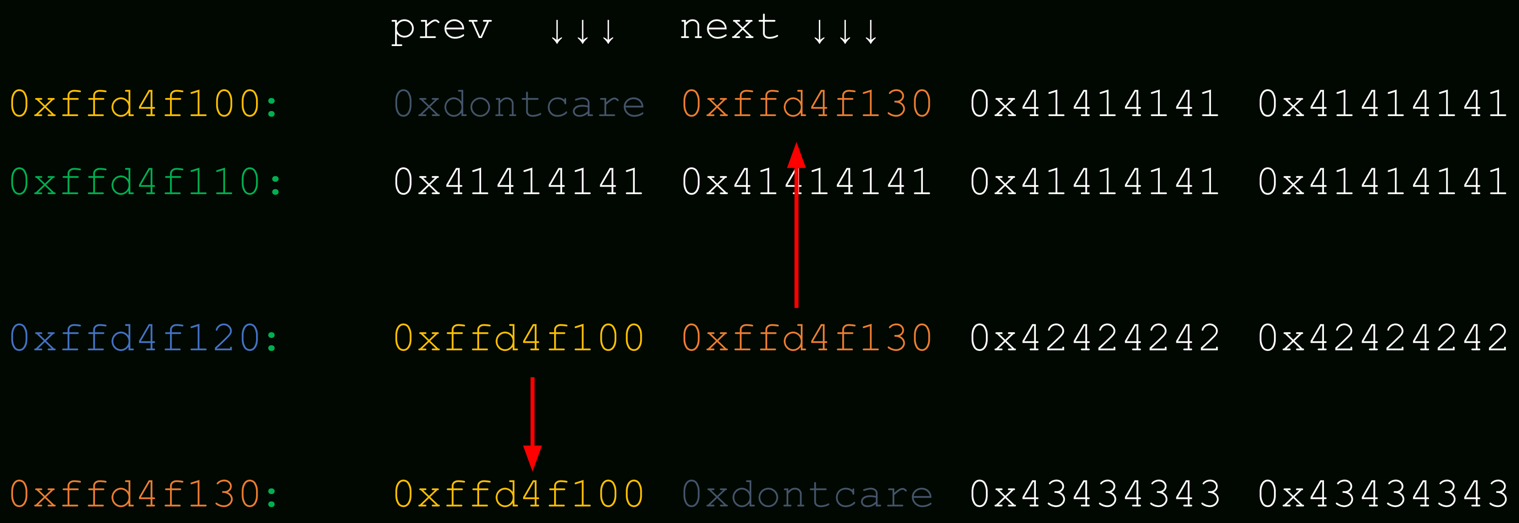
```
                 prev  ↓↓↓  next ↓↓↓
0xffd4f100:      0xdontcare 0xffd4f120 0x41414141 0x41414141
0xffd4f110:      0x41414141 0x41414141 0x41414141 0x41414141

0xffd4f120:      0xffd4f100 0xffd4f130 0x42424242 0x42424242

0xffd4f130:      0xffd4f120 0xdontcare 0x43434343 0x43434343
```

What happens when we free(0xffd4f120)?

# unlink() sploit'n



```
                prev  ↓↓↓   next ↓↓↓
0xffd4f100:     0xdontcare  0xffd4f130  0x41414141  0x41414141
0xffd4f110:     0x41414141  0x41414141  0x41414141  0x41414141

0xffd4f120:     0xffd4f100  0xffd4f130  0x42424242  0x42424242

0xffd4f130:     0xffd4f100  0xdontcare  0x43434343  0x43434343
```

What happens when we free(0xffd4f120)?

0xffd4f100's next now points to 0xffd4f130

0xffd4f130's prev now points to 0xffd4f100

# unlink() sploit’n

```
               prev  ↓↓↓   next ↓↓↓
0xffd4f100:    0xdontcare 0xffd4f130 0x41414141 0x41414141
0xffd4f110:    0x41414141 0x41414141 0x41414141 0x41414141

0xffd4f120:    0xdontcare 0xdontcare 0xdontcare 0xdontcare

0xffd4f130:    0xffd4f100 0xdontcare 0x43434343 0x43434343
```

What happens when we free(0xffd4f120)?

0xffd4f100’s next now points to 0xffd4f130

0xffd4f130’s prev now points to 0xffd4f100

# unlink() sploit'n

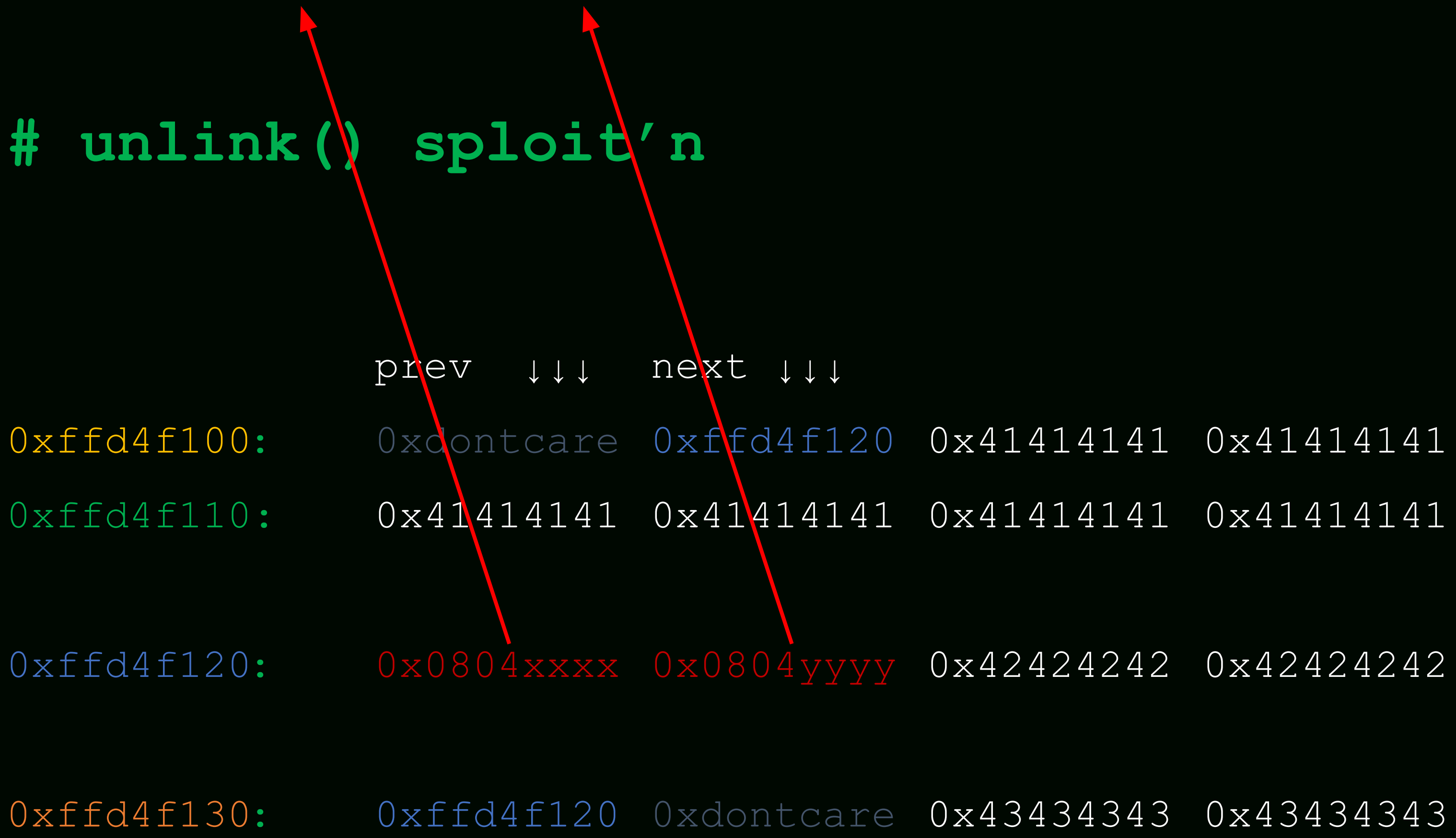
```
                prev  ↓↓↓   next ↓↓↓
0xffd4f100:     0xdontcare  0xffd4f120  0x41414141  0x41414141
0xffd4f110:     0x41414141  0x41414141  0x41414141  0x41414141

0xffd4f120:     0x0804xxxx  0x0804yyyy  0x42424242  0x42424242

0xffd4f130:     0xffd4f120  0xdontcare  0x43434343  0x43434343
```

What happens when we free(0xffd4f120)?

# unlink() sploit'n



```
                 prev  ↓↓↓   next ↓↓↓
0xffd4f100:     0xdontcare 0xffd4f120 0x41414141 0x41414141
0xffd4f110:     0x41414141 0x41414141 0x41414141 0x41414141

0xffd4f120:     0x0804xxxx 0x0804yyyy 0x42424242 0x42424242

0xffd4f130:     0xffd4f120 0xdontcare 0x43434343 0x43434343
```

What happens when we free(0xffd4f120)?

0xffd4f100 and 0xffd4f130 untouched.

0x0804xxxx now point to the heap.

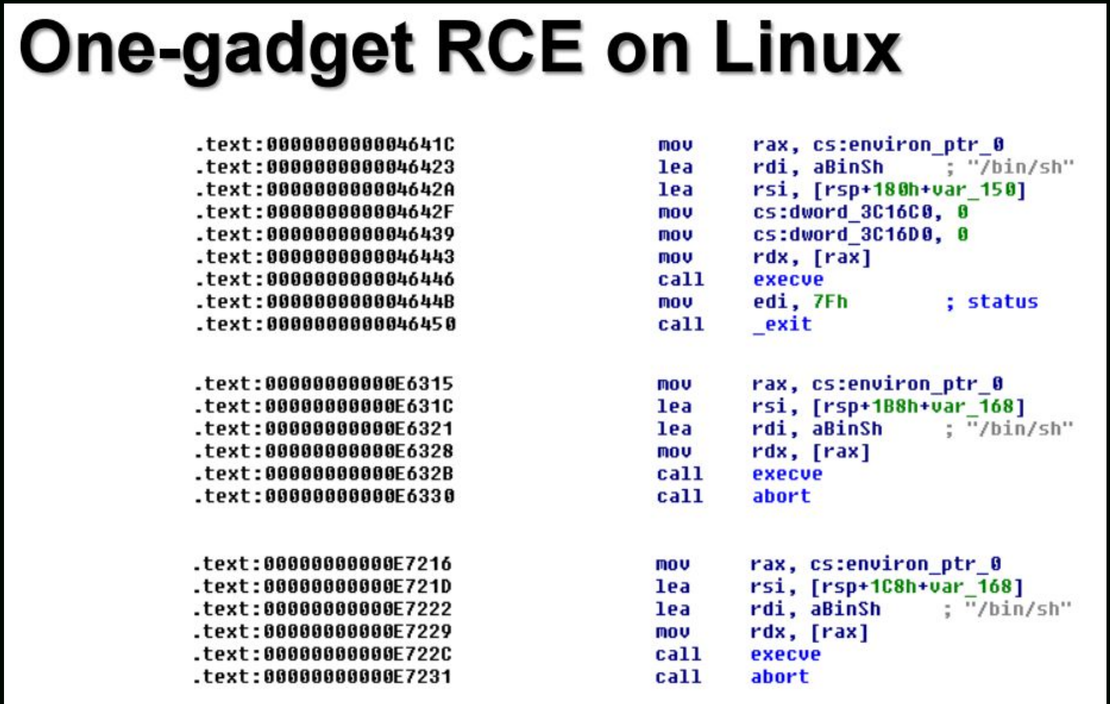
# good targets

Attacker can choose what to overwrite

- If no NX/DEP, can point GOT/DTORS to heap for $hellcode
- Else, point to some useful code..?
- Heap pointers, etc, for structs on the heap
  Quite useful
  Recently OS X got rooted with this
- Else, point to pivot for ROP & hope......? :(

# good targets

```
void MLG_360_noscope()   in libc (64 bit):
```



One-gadget RCE on Linux

```
.text:000000000004641C                mov     rax, cs:environ_ptr_0
.text:0000000000046423                lea     rdi, aBinSh     ; "/bin/sh"
.text:000000000004642A                lea     rsi, [rsp+180h+var_150]
.text:000000000004642F                mov     cs:dword_3C16C0, 0
.text:0000000000046439                mov     cs:dword_3C16D0, 0
.text:0000000000046443                mov     rdx, [rax]
.text:0000000000046446                call    execve
.text:000000000004644B                mov     edi, 7Fh        ; status
.text:0000000000046450                call    _exit

.text:00000000000E6315                mov     rax, cs:environ_ptr_0
.text:00000000000E631C                lea     rsi, [rsp+1B8h+var_168]
.text:00000000000E6321                lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000E6328                mov     rdx, [rax]
.text:00000000000E632B                call    execve
.text:00000000000E6330                call    abort

.text:00000000000E7216                mov     rax, cs:environ_ptr_0
.text:00000000000E721D                lea     rsi, [rsp+1C8h+var_168]
.text:00000000000E7222                lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000E7229                mov     rdx, [rax]
.text:00000000000E722C                call    execve
.text:00000000000E7231                call    abort
```

# some kinds of vulns

- **Heap overflow**

  ```
  strcpy(malloc(16), user_input);
  ```

- **Use-after-free**

  ```
  free(struct);    // Let’s say struct has pointers
  alloc_some_memory();
  keep_using(struct);
  ```

- **Type confusion**

  Either real bug like shitty casting (uninteresting).
  Or induced by the attacker through OF or UAF or...

- **Others**

# approaches

- **Info leak**

  Find location of heap (under ASLR).
  Try to disclose adjacent or old memory contents:

  ```
  char *p = malloc(16);
  strncpy(p, user_input, 16);
  printf(“%s”, p);

  > “AAAABBBBCCCCDDDD\xA0\xBF\x04\x06”
  ```

  From here, do some maths to deduce heap address.

- **Heap spraying**

  Allocating many fixed-size objects.
  Idea is to make heap uniform (predictable).

  Usually followed by corrupting an object and hoping.

We'll
Do It
Live!
DEMO GODS
PLEASE LET THIS DEMO WORK
memegenerator.net

# glibc impl

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

# glibc impl

- Some heap overflows can be exploited relatively easily
  (see sploitfun links at the end)
  fastbins demo?

- If you are constrained in your allocs/deallocs,
  or cannot control data near where you wanna write
  you’re gonna have a bad time

- Glibc 2.19 (Ubuntu 14.04?) -> 2.21 (current)
  idk if they are changing much (allegedly hardened it)

- It’s fukt. See sploitfun for 2 working types of corruption.

# links

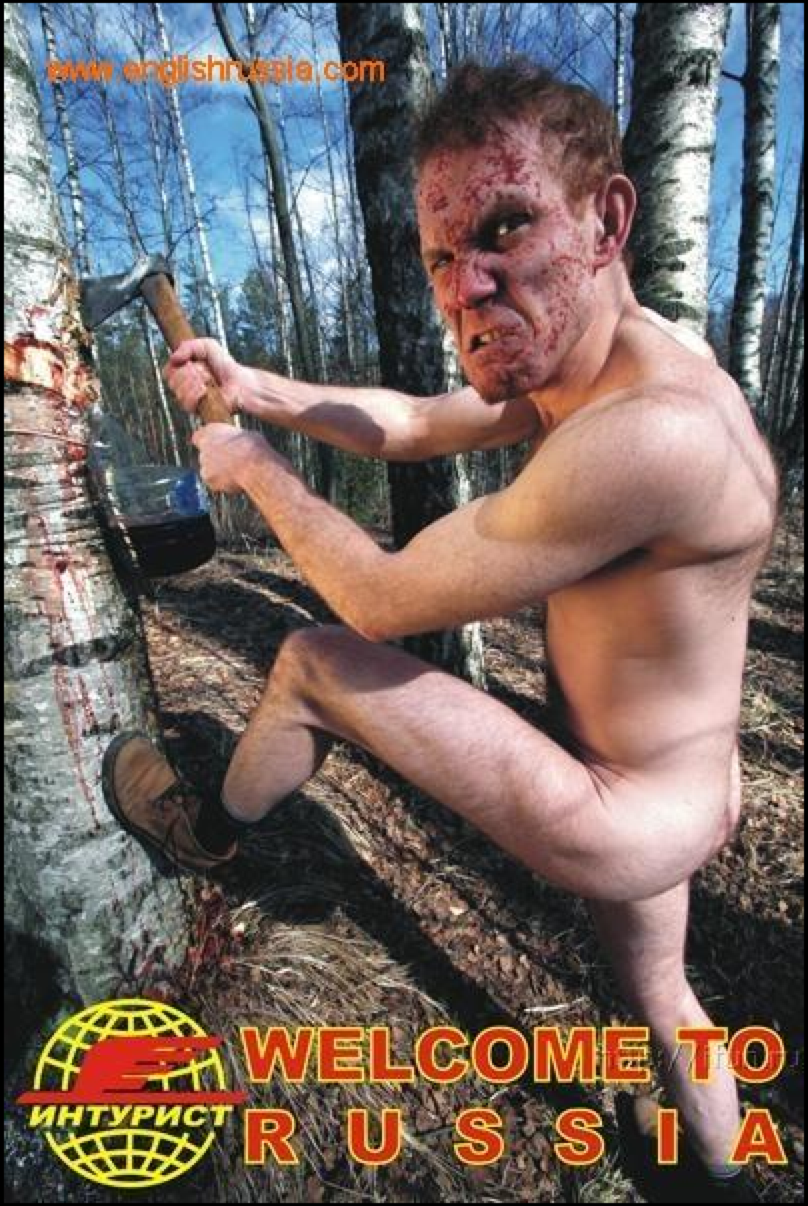
- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
  Overview of how glibc malloc works.
  Recent.
  Disclaimer: TLDR.

- https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/
  Overview of glibc-malloc sploit'n possibilities.
  Includes POCs.

  Recent despite being based on 2005's Malloc Maleficarum.
  Good read.
  Explains glibc malloc patches.

- http://code.woboq.org/userspace/glibc/malloc/malloc.c.html
  Malloc source (glibc 2.21 ?)

- http://googleprojectzero.blogspot.com.au/


www.englishrussia.com
ИНТУРИСТ
WELCOME TO
RUSSIA