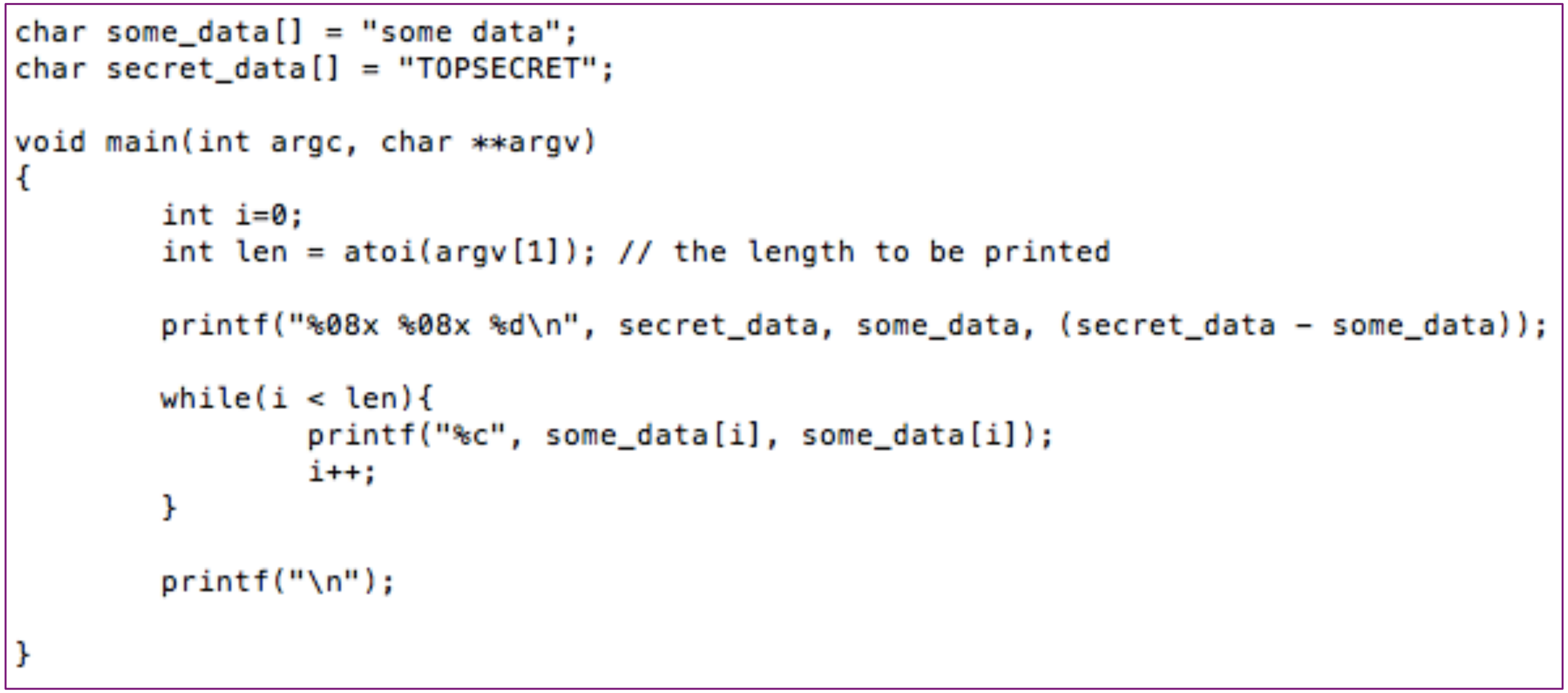
More Vulnerabilities
(buffer overreads, format string, integer overflow, heap overflows)

Chester Rebeiro

Indian Institute of Technology Madras

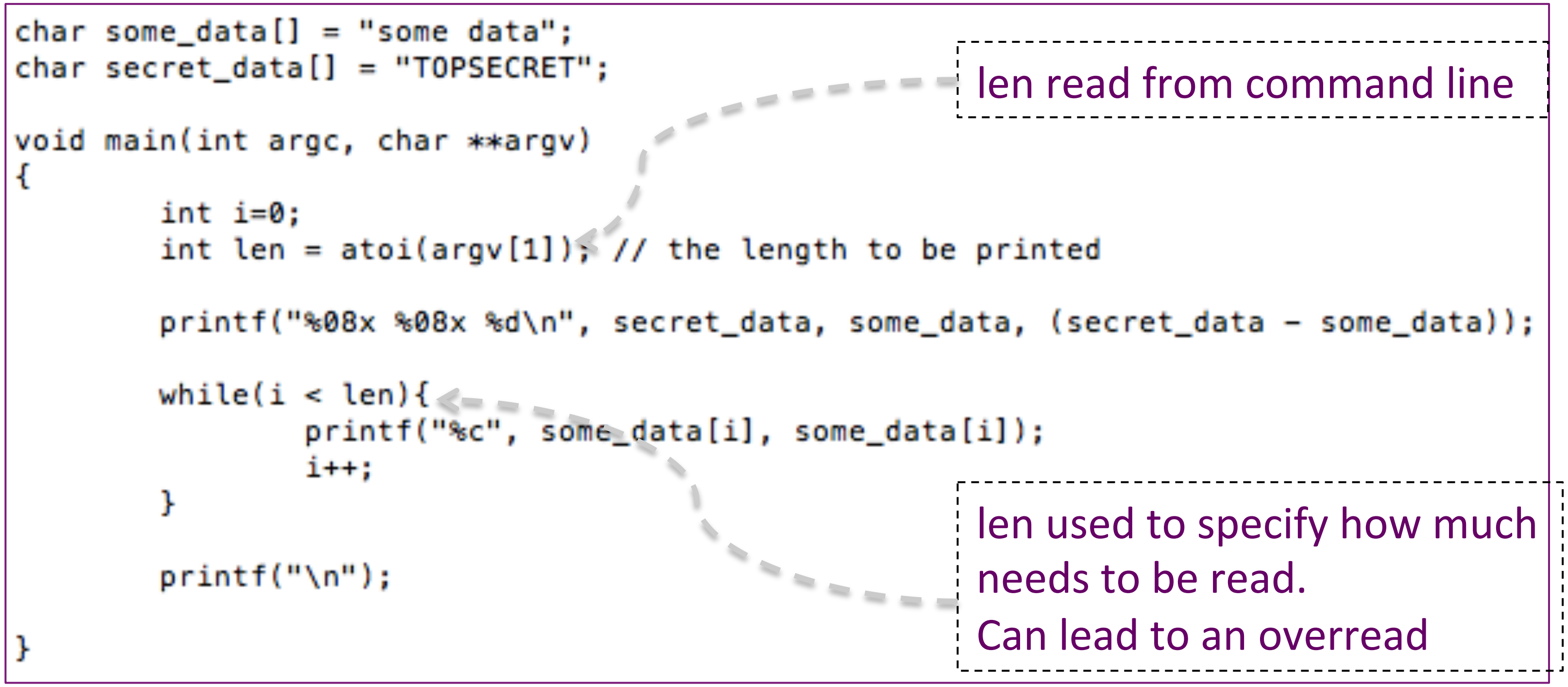
Buffer Overreads

Buffer Overread Example



```
char some_data[] = "some data";
char secret_data[] = "TOPSECRET";

void main(int argc, char **argv)
{
        int i=0;
        int len = atoi(argv[1]); // the length to be printed

        printf("%08x %08x %d\n", secret_data, some_data, (secret_data - some_data));

        while(i < len){
                printf("%c", some_data[i], some_data[i]);
                i++;
        }

        printf("\n");

}
```

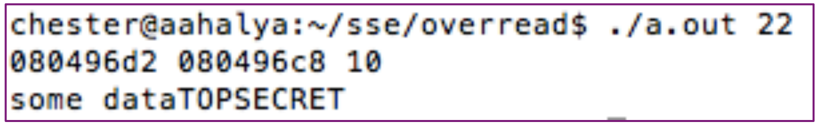
Buffer Overread Example



```
char some_data[] = "some data";
char secret_data[] = "TOPSECRET";

void main(int argc, char **argv)
{
        int i=0;
        int len = atoi(argv[1]); // the length to be printed

        printf("%08x %08x %d\n", secret_data, some_data, (secret_data - some_data));

        while(i < len){
                printf("%c", some_data[i], some_data[i]);
                i++;
        }

        printf("\n");

}
```

len read from command line

len used to specify how much needs to be read.
Can lead to an overread



```
chester@aahalya:~/sse/overread$ ./a.out 22
080496d2 080496c8 10
some dataTOPSECRET
```

Buffer Overreads

- Cannot be prevented by canaries

  canaries only look for changes

- Cannot be prevented by the W^X bit

  we are not executing any code

- Cannot be prevented by ASLR

  not moving out of the segment

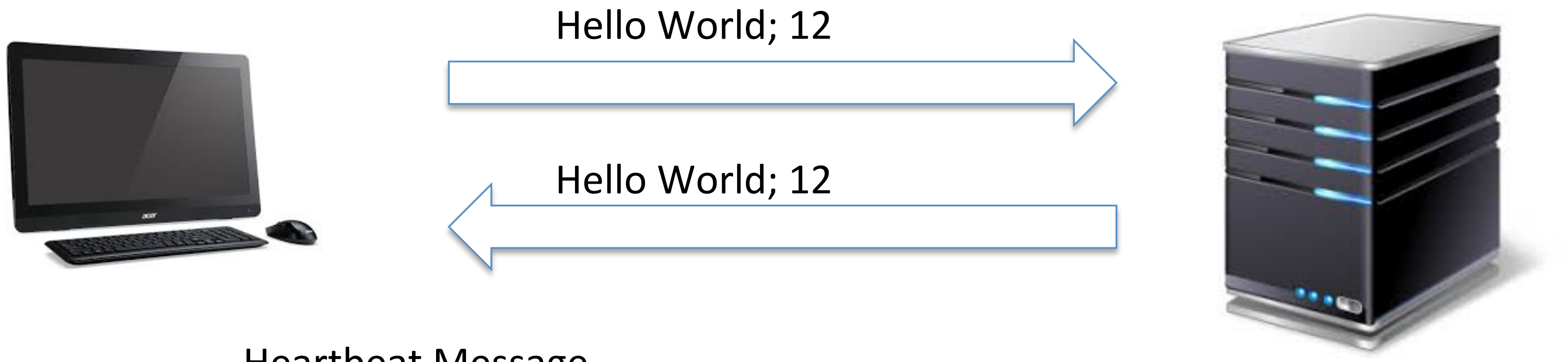
- Can be prevented by compiler and hardware level changes

Heartbleed : A buffer overread malware

- 2012 – 2014
  - Introduced in 2012; disclosed in 2014
- CVE-2014-0160
- Target : OpenSSL implementation of TLS – transport layer security
  - TLS defines crypto-protocols for secure communication
  - Used in applications such as email, web-browsing, VoIP, instant messaging,
  - Provide privacy and data integrity

https://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Heartbeat



Hello World; 12

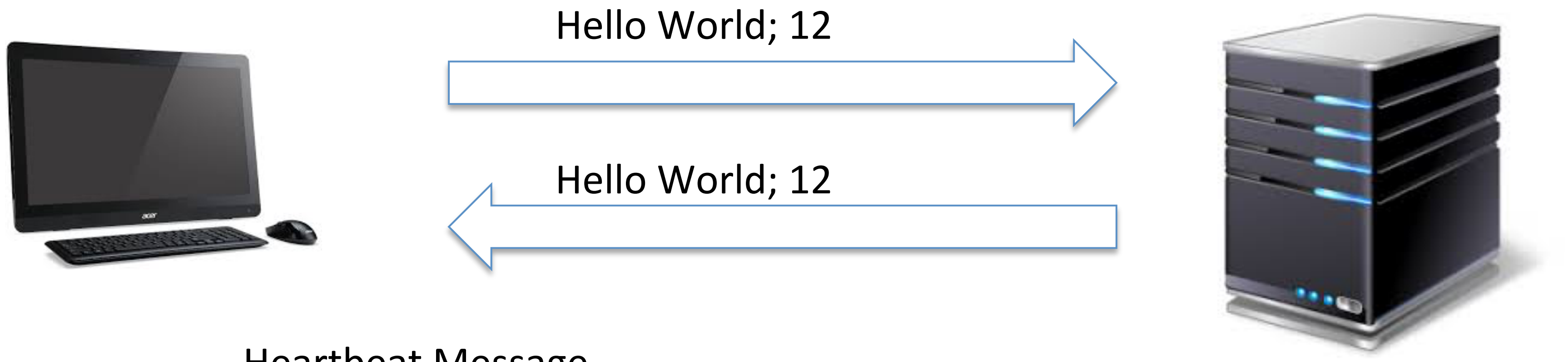
Hello World; 12

Heartbeat Message

type	length	payload	padding

- A component of TLS that provides a means to keep alive secure communication links
  - This avoids closure of connections due to some firewalls
  - Also ensures that the peer is still alive

Heartbeat



Heartbeat Message

type	length	payload	padding

- Client sends a heart beat message with some payload
- Server replies with the same payload to signal that everything is OK

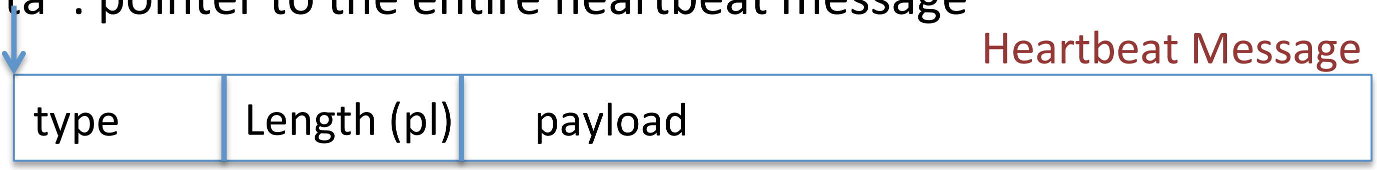
SSL3 struct and Heartbeat

- Heartbeat message arrives via an SSL3 structure, which is defined as follows

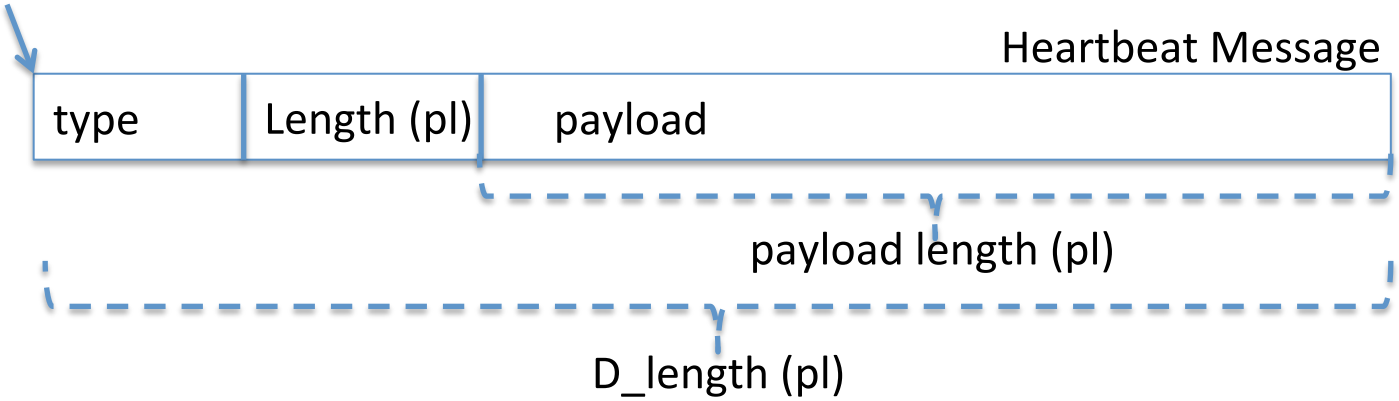
```
struct ssl3_record_st
{
  unsigned int D_length;    /* How many bytes available */
  [...]
  unsigned char *data;    /* pointer to the record data */
  [...]
} SSL3_RECORD;
```

length : length of the heartbeat message

data : pointer to the entire heartbeat message



Payload and Heartbeat length



- ***payload_length***: controlled by the heartbeat message creator
  - Can never be larger than D_length
  - However, this check was never done!!!
    - Thus allowing the heartbeat message creator to place some arbitrary large number in the payload_length
    - Resulting in overread

Overread Example

Heartbeat sent to victim

SSLv3 record:

Length
4 bytes

HeartbeatMessage:

Type	Length	Payload data	
TLS1_HB_REQUEST	65535 bytes	1 byte	

Attacker sends a heartbeat message with a single byte payload to the server. However, the pl_length is set to 65535 (the max permissible pl_length)

Victim's response

SSLv3 record:

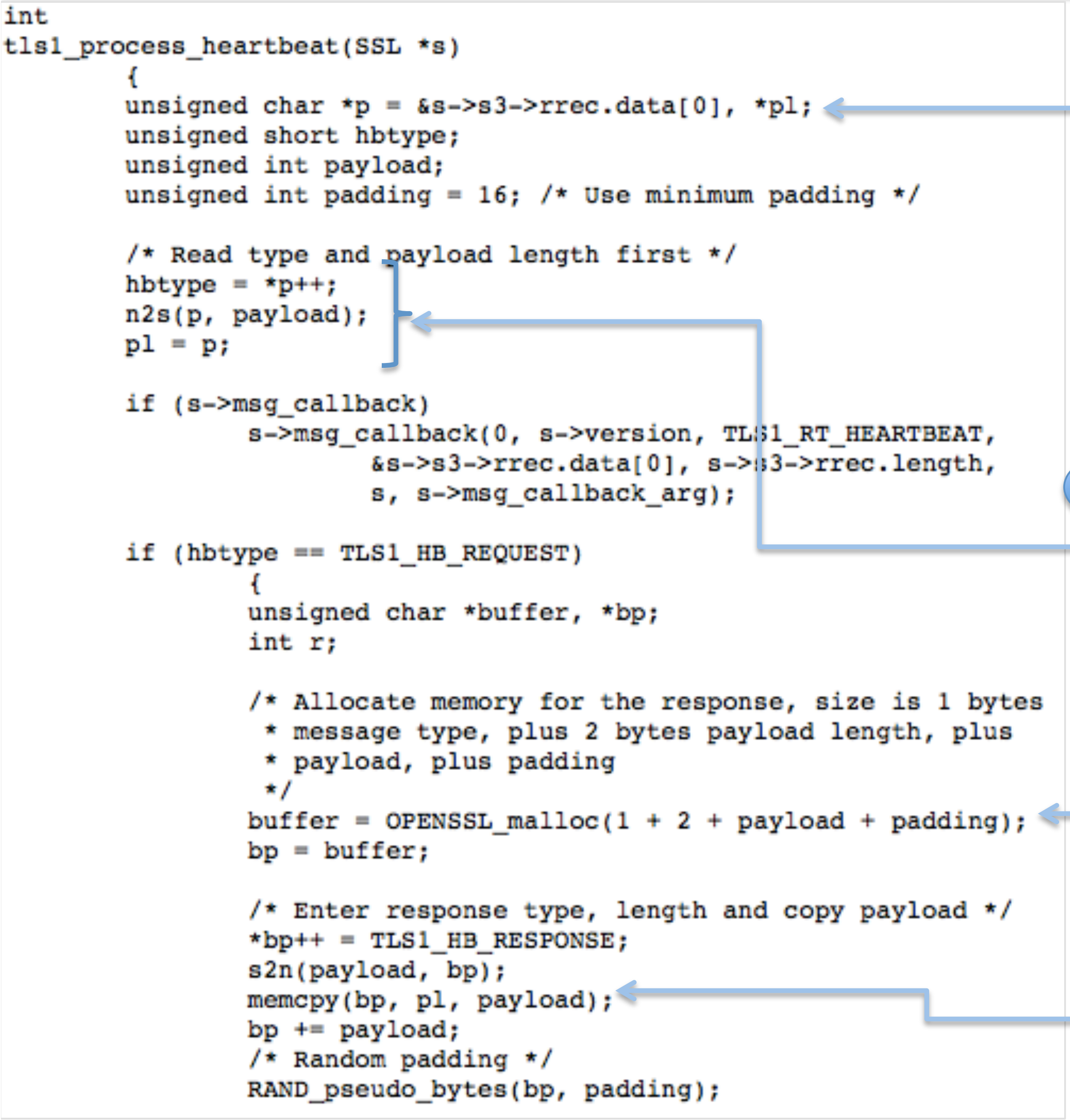
Length
65538 bytes

HeartbeatMessage:

Type	Length	Payload data	
TLS1_HB_RESPONSE	65535 bytes	65535 bytes	

Victim ignores the SSL3 length (of 4 bytes), Looks only at the pl_length and returns a payload of 65535 bytes. In the payload, only 1 byte is victim's data remaining 65534 from its own memory space.

Broken OpenSSL code@victim



```
int
tls1_process_heartbeat(SSL *s)
        {
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        unsigned short hbtype;
        unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */

        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;

        if (s->msg_callback)
                s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

        if (hbtype == TLS1_HB_REQUEST)
                {
                unsigned char *buffer, *bp;
                int r;

                /* Allocate memory for the response, size is 1 bytes
                 * message type, plus 2 bytes payload length, plus
                 * payload, plus padding
                 */
                buffer = OPENSSL_malloc(1 + 2 + payload + padding);
                bp = buffer;

                /* Enter response type, length and copy payload */
                *bp++ = TLS1_HB_RESPONSE;
                s2n(payload, bp);
                memcpy(bp, pl, payload);
                bp += payload;
                /* Random padding */
                RAND_pseudo_bytes(bp, padding);
```

1 p points to the attackers heart beat packet which the victim just received.

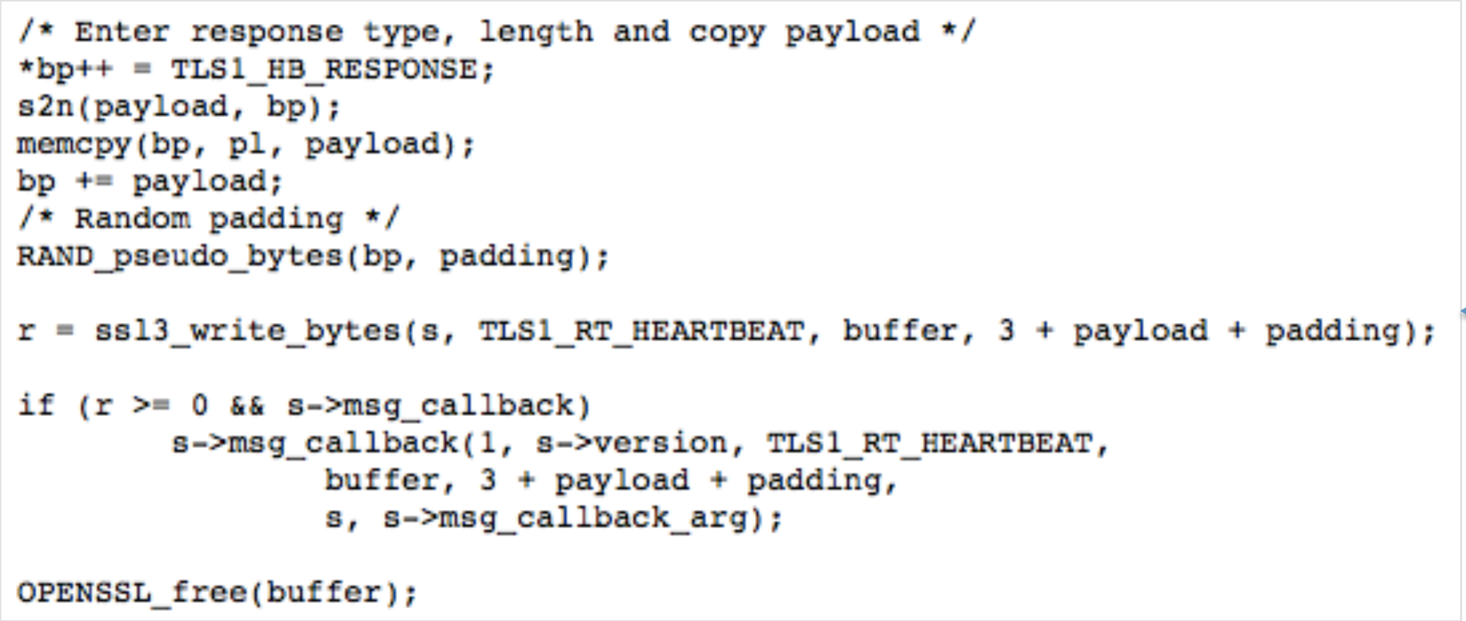
2 get the heartbeat type;
fill payload with size of payload (pl in our notation)
This is picked up from the attackers payload and contains 65535

3 Allocate buffer of 3 + 65535 + 16 bytes

4 memcpy grossly overreads from the victim's heap

https://git.openssl.org/gitweb/?p=openssl.git;a=blob;f=ssl/t1_lib.c;h=a2e2475d136f33fa26958fd192b8ace158c4899d#l3969

Broken OpenSSL code@victim

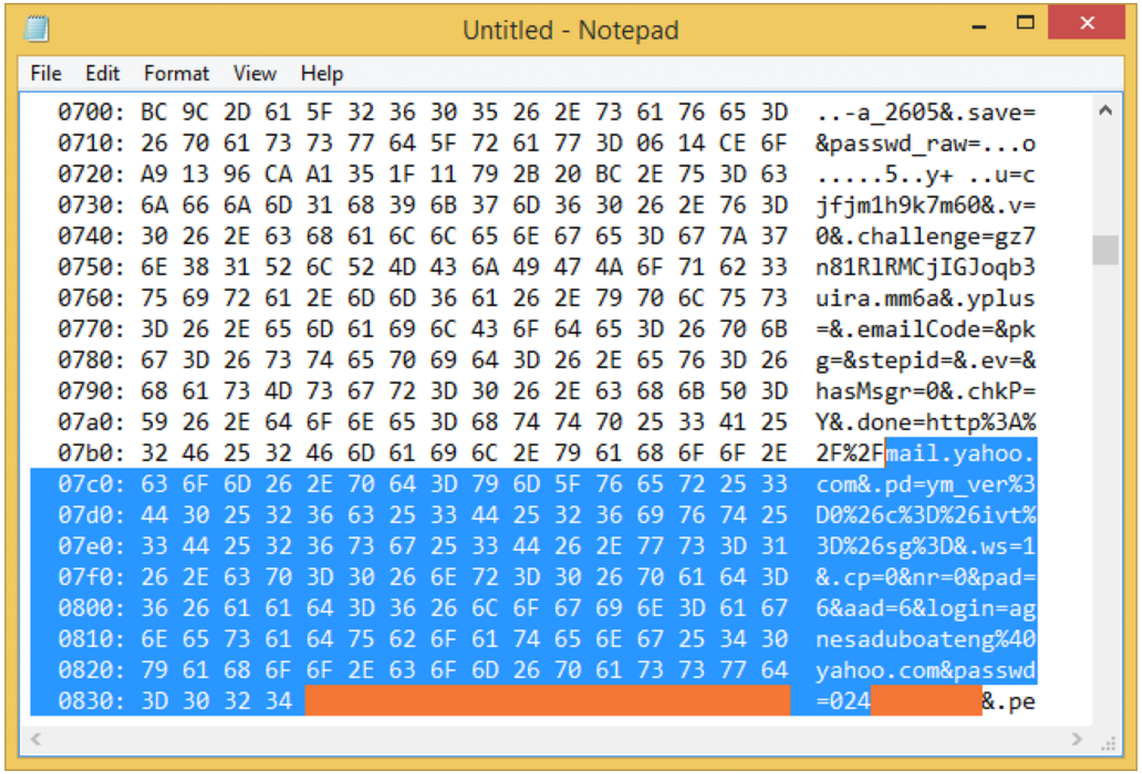


```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

if (r >= 0 && s->msg_callback)
        s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
                buffer, 3 + payload + padding,
                s, s->msg_callback_arg);

OPENSSL_free(buffer);
```

5

Add padding and send the response heartbeat message back to the attacker

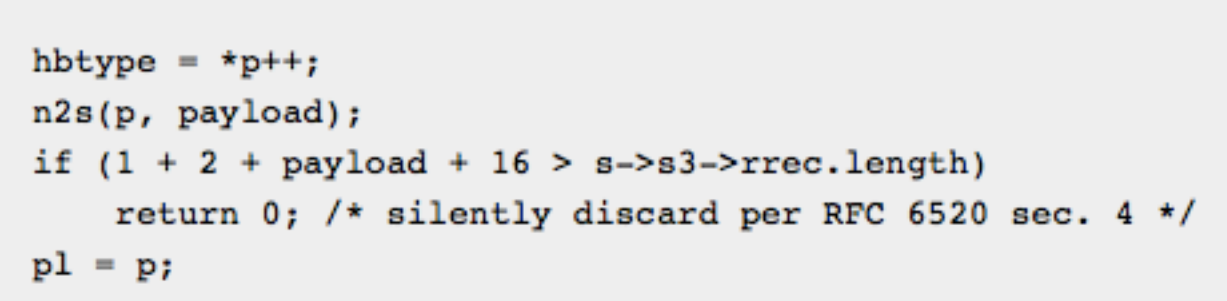
65534 byte return payload may contain sensitive data



```
0700: BC 9C 2D 61 5F 32 36 30 35 26 2E 73 61 76 65 3D  ..-a_2605&.save=
0710: 26 70 61 73 73 77 64 5F 72 61 77 3D 06 14 CE 6F  &passwd_raw=...o
0720: A9 13 96 CA A1 35 1F 11 79 2B 20 BC 2E 75 3D 63  .....5..y+ ..u=c
0730: 6A 66 6A 6D 31 68 39 6B 37 6D 36 30 26 2E 76 3D  jfjm1h9k7m60&.v=
0740: 30 26 2E 63 68 61 6C 6C 65 6E 67 65 3D 67 7A 37  0&.challenge=gz7
0750: 6E 38 31 52 6C 52 4D 43 6A 49 47 4A 6F 71 62 33  n81RlRMCjIGJoqb3
0760: 75 69 72 61 2E 6D 6D 36 61 26 2E 79 70 6C 75 73  uira.mm6a&.yplus
0770: 3D 26 2E 65 6D 61 69 6C 43 6F 64 65 3D 26 70 6B  =&.emailCode=&pk
0780: 67 3D 26 73 74 65 70 69 64 3D 26 2E 65 76 3D 26  g=&stepid=&.ev=&
0790: 68 61 73 4D 73 67 72 3D 30 26 2E 63 68 6B 50 3D  hasMsgr=0&.chkP=
07a0: 59 26 2E 64 6F 6E 65 3D 68 74 74 70 25 33 41 25  Y&.done=http%3A%
07b0: 32 46 25 32 46 6D 61 69 6C 2E 79 61 68 6F 6F 2E  2F%2Fmail.yahoo.
07c0: 63 6F 6D 26 2E 70 64 3D 79 6D 5F 76 65 72 25 33  com&.pd=ym_ver%3
07d0: 44 30 25 32 36 63 25 33 44 25 32 36 69 76 74 25  D0%26c%3D%26ivt%
07e0: 33 44 25 32 36 73 67 25 33 44 26 2E 77 73 3D 31  3D%26sg%3D&.ws=1
07f0: 26 2E 63 70 3D 30 26 6E 72 3D 30 26 70 61 64 3D  &.cp=0&nr=0&pad=
0800: 36 26 61 61 64 3D 36 26 6C 6F 67 69 6E 3D 61 67  6&aad=6&login=ag
0810: 6E 65 73 61 64 75 62 6F 61 74 65 6E 67 25 34 30  nesaduboateng%40
0820: 79 61 68 6F 6F 2E 63 6F 6D 26 70 61 73 73 77 64  yahoo.com&passwd
0830: 3D 30 32 34 [redacted]                            =024[redacted]&.pe
```

Further, invocations of similar false heartbleed will result in another 64KB of the heap to be read.
In this way, the attacker can scrape through the victim's heap.

The patch in OpenSSL



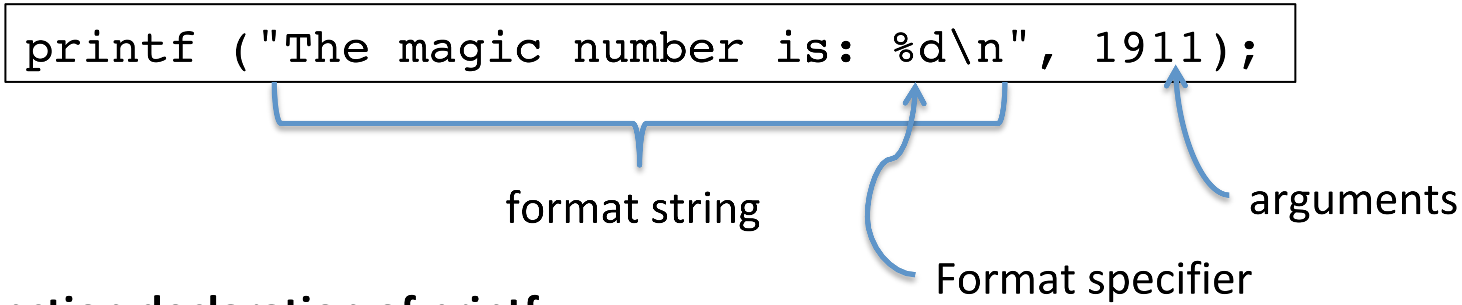
```
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

Discard the heartbeat response if it happens to be greater than the length in the SSL3 structure (i.e. D_length)

Format String Vulnerabilities

https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Format Strings



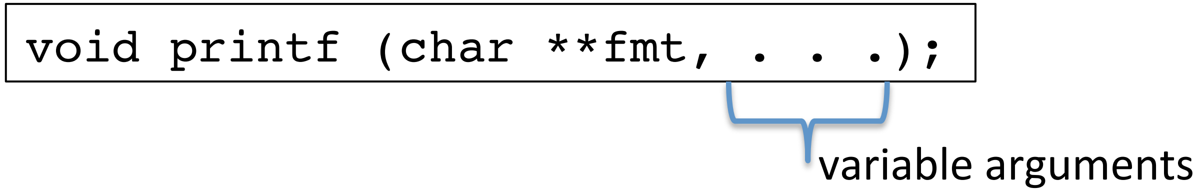
```
printf ("The magic number is: %d\n", 1911);
```

format string

arguments

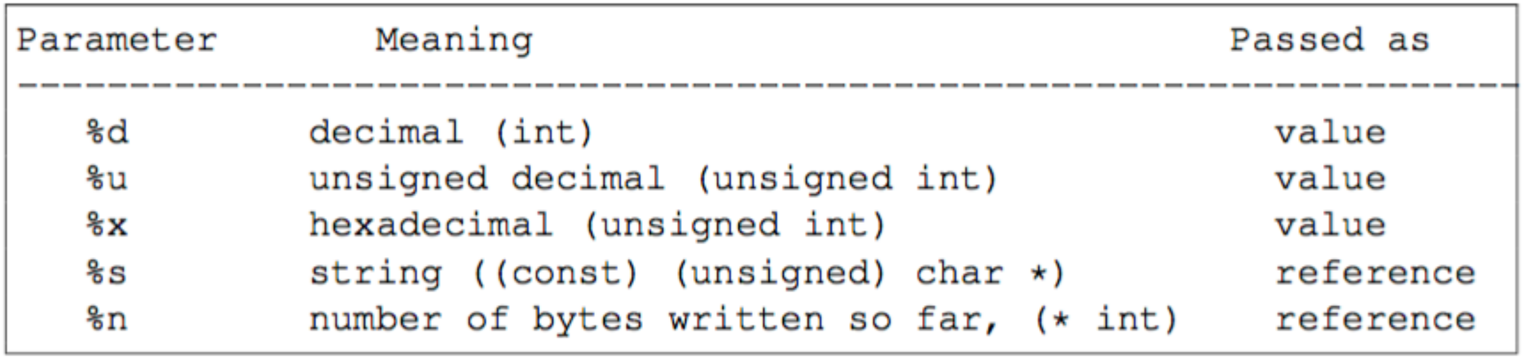
Format specifier

Function declaration of printf



```
void printf (char **fmt, . . .);
```

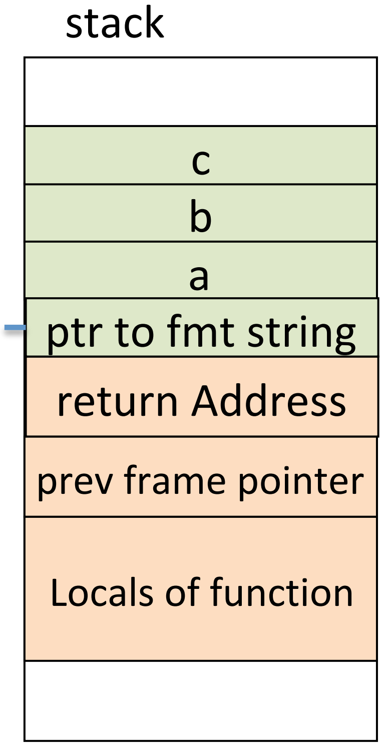
variable arguments



Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

printf invocation

```
void main(){
    printf (“%d %d %d\n", a, b, c);
}
```



stack

c

b

a

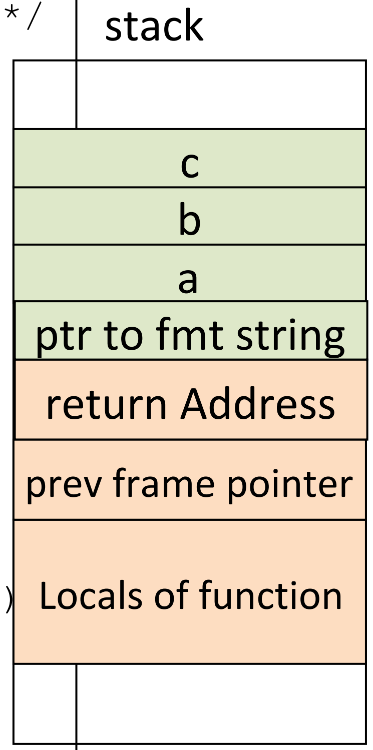
ptr to fmt string

return Address

prev frame pointer

Locals of function

```
void printf(char *fmt, ...){
   va_list ap; /* points to each unnamed arg in turn */
   char *p, *sval;
   int ival;
   double dval;
   va_start(ap, fmt); /*make ap point to 1st unnamed arg */
   for (p = fmt; *p; p++) {
      if (*p != '%’) {
         putchar(*p);
         continue;
      }
      switch (*++p) {
         case 'd':
            ival = va_arg(ap, int);
            print_int(ival);
            break;
           |  | |  |  |
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
      }
   }
   va_end(ap); /* clean up when done */
}
```

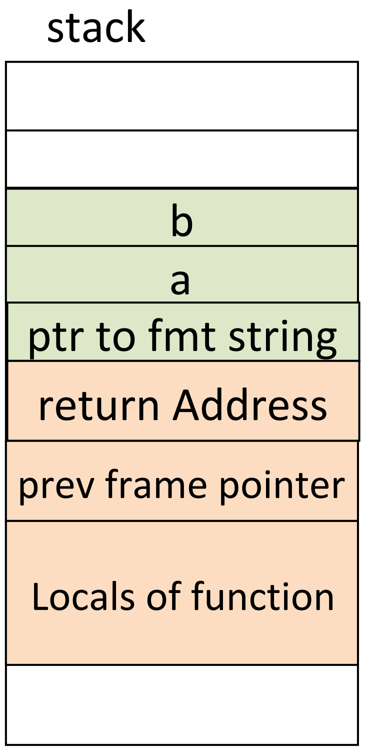


Insufficient Arguments to printf

```
void main(){
   printf (“%d %d %d\n", a, b);
}
```

3 format specifiers

But only 2 arguments



stack

b

a

ptr to fmt string

return Address

prev frame pointer

Locals of function

Can the compiler detect this inconsistency

- Generally does not
- Would need internal details of printf, making the compiler library dependent.
- Format string may be created at runtime

Can the printf function detect this inconsistency

- Not easy
- Just picks out arguments from the stack, whenever it sees a format specifier

Exploiting inconsistent printf

- Crashing a program

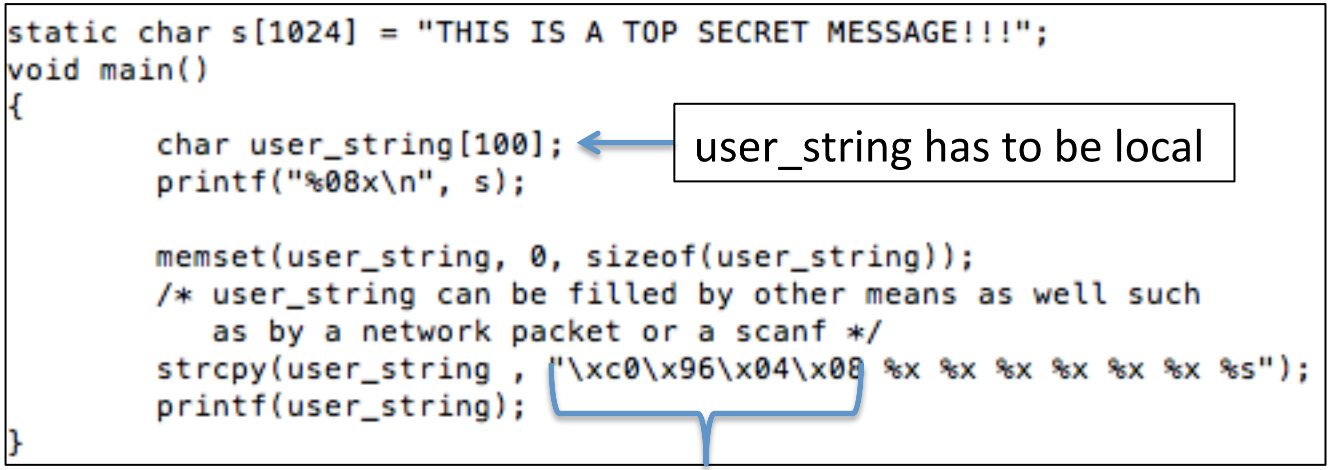
```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

- Printing contents of the stack

```
printf ("%x %x %x %x");
```

Exploiting inconsistent printf

- Printing any memory location



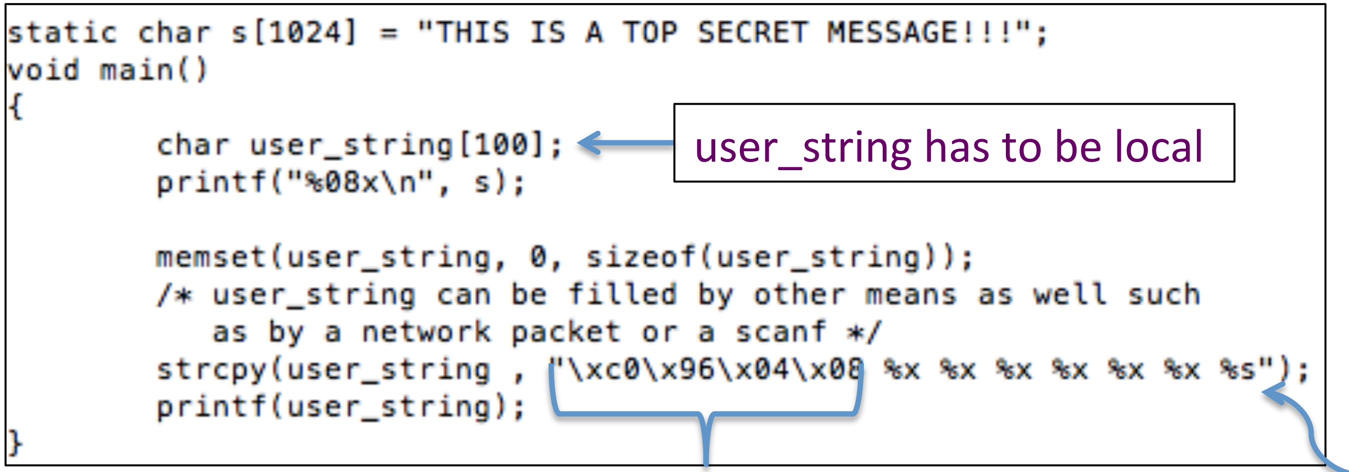
```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xc0\x96\x04\x08 %x %x %x %x %x %x %s");
        printf(user_string);
}
```

user_string has to be local

This should have the contents of s

Exploiting inconsistent printf

- Printing any memory location

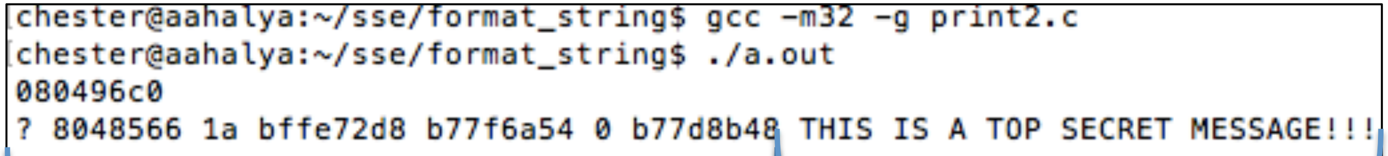


```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xc0\x96\x04\x08 %x %x %x %x %x %x %s");
        printf(user_string);
}
```

user_string has to be local

%s, picks pointer from the stack and prints from the pointer till \0

This should have the contents of s

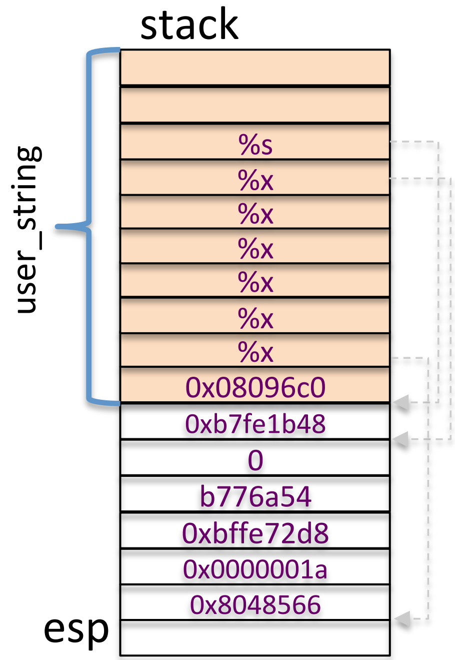


```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

contents of the stack printed by the 6 %x

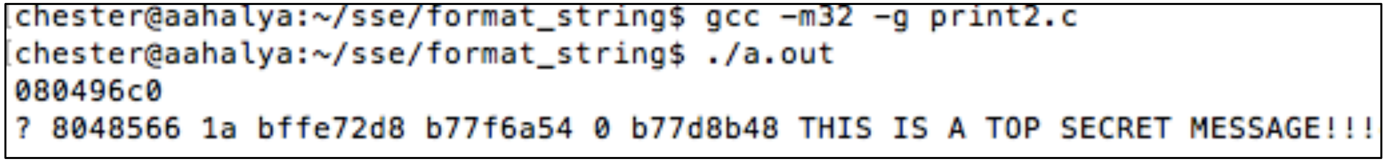
string pointed to by 0x080496c0. this happens to be ‘s’

Digging deeper



```
printf(user_string);
```

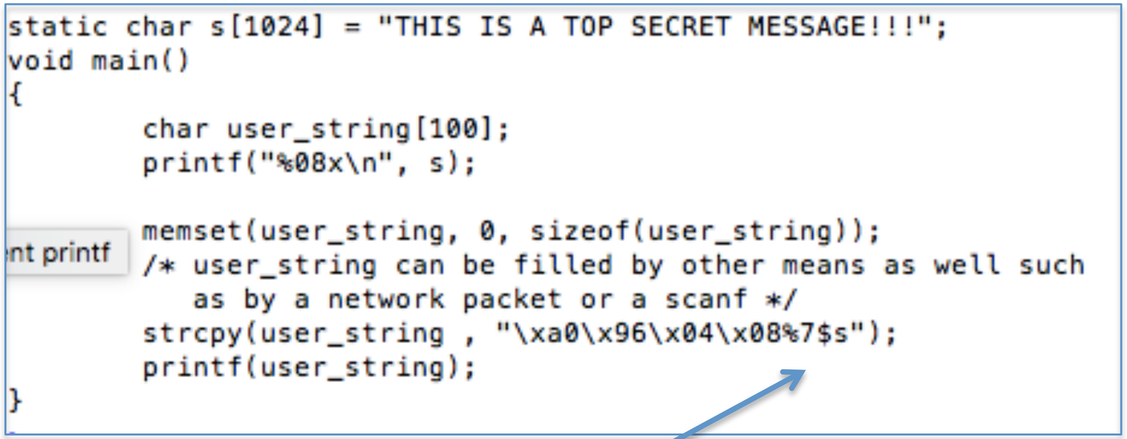
- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
  - It reads the argument from the stack
  - and increments the va_arg pointer
- If we have sufficient %x's, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0



```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

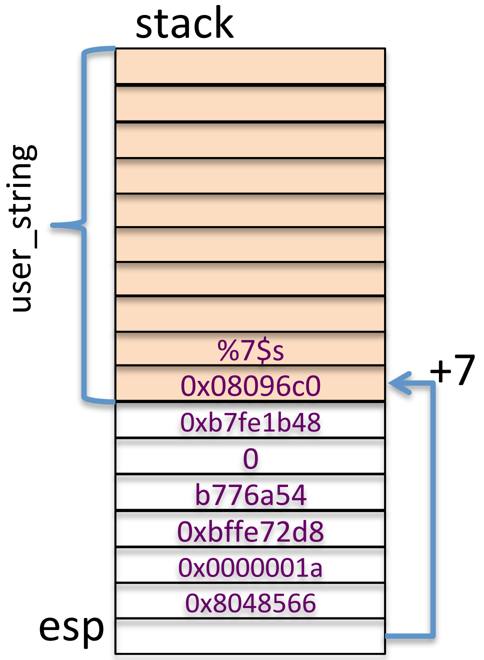
More Format Specifiers

- Reduce the number of %x with %N$s



```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

nt printf  memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xa0\x96\x04\x08%7$s");
        printf(user_string);
}
```

Pick the 7th argument from the stack.



Overwrite
an arbitrary location

%n format specifier : returns the number of characters printed so far.

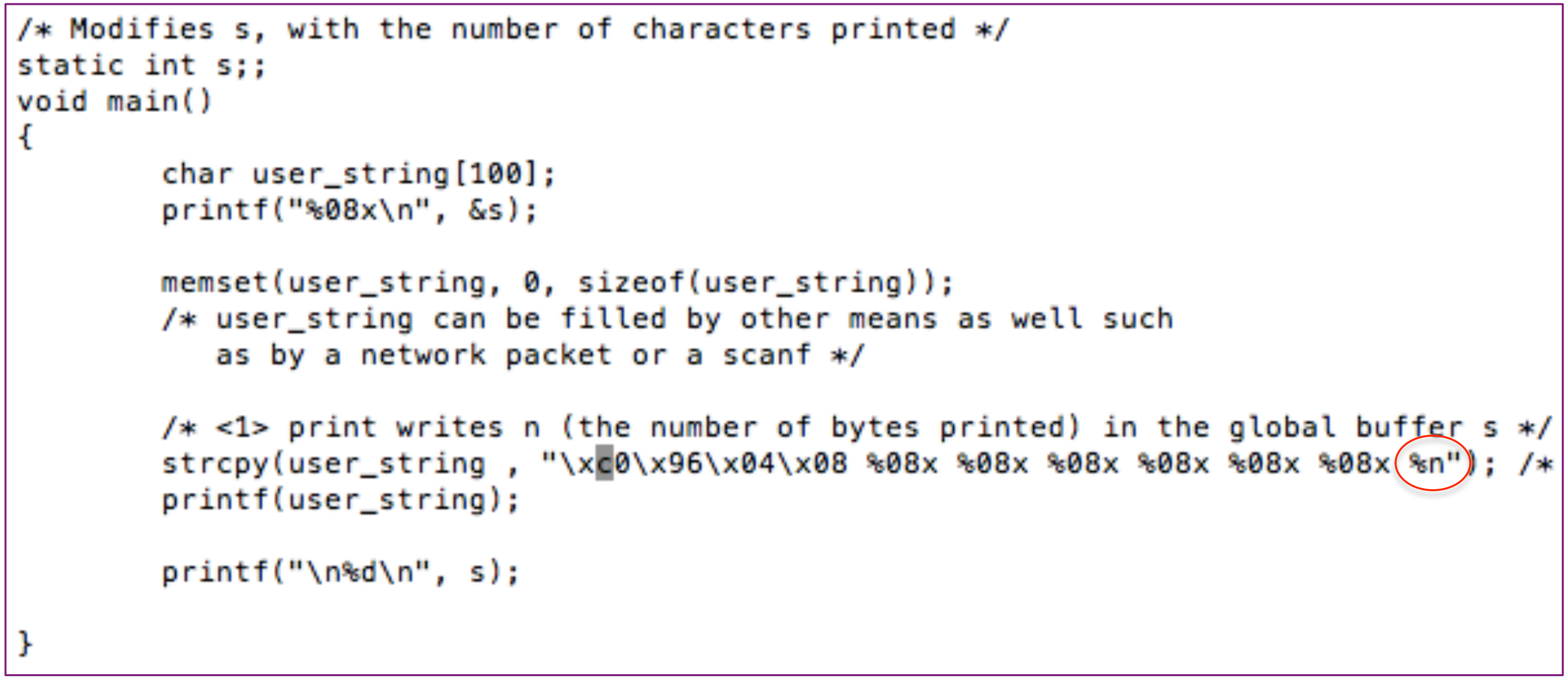
- ‘i’ is filled with 5 here

```
int i;
printf(“12345%n”, &i);
```

Using the same approach to read data from any location, printf can be used to modify a location as well

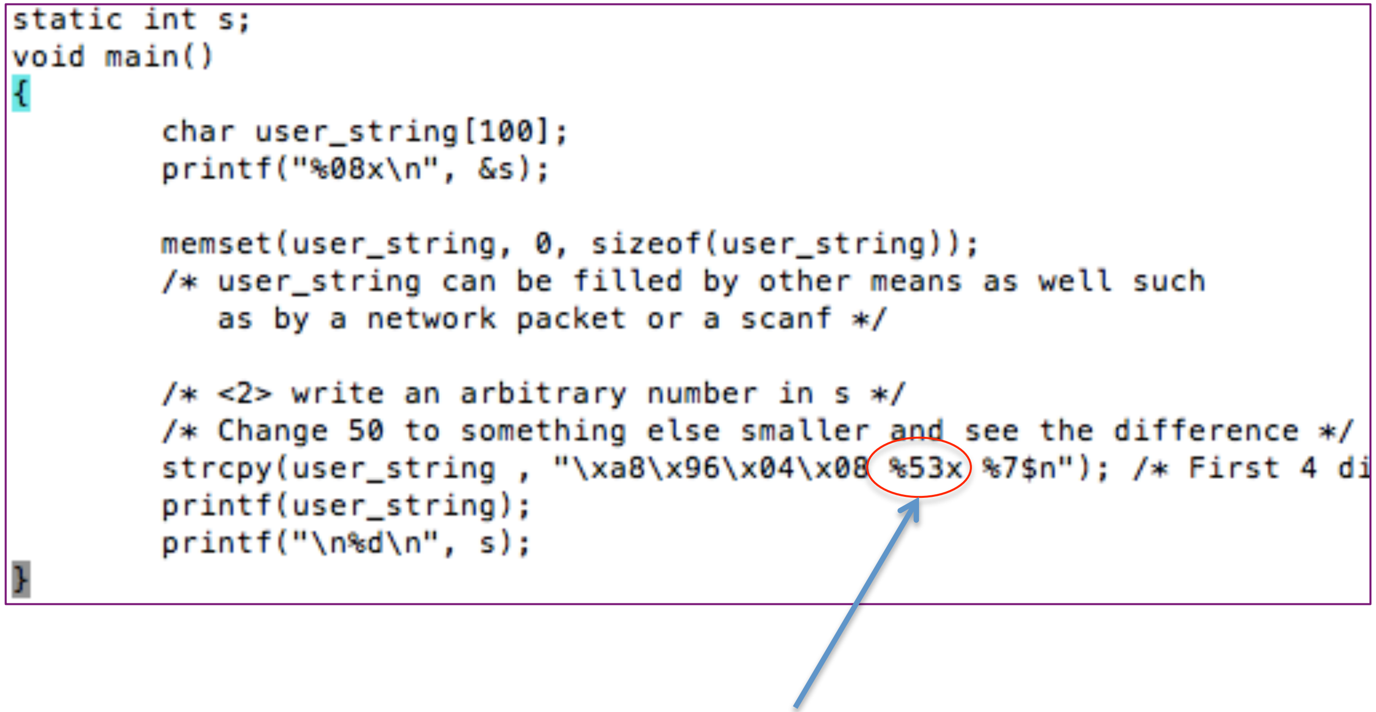
Can be used to change function pointers as well as return addresses

Overwrite Arbitrary Location with some number



```
/* Modifies s, with the number of characters printed */
static int s;;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */

        /* <1> print writes n (the number of bytes printed) in the global buffer s */
        strcpy(user_string , "\xc0\x96\x04\x08 %08x %08x %08x %08x %08x %08x %n"); /*
        printf(user_string);

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

}
```

Overwrite Arbitrary Location with Arbitrary Number

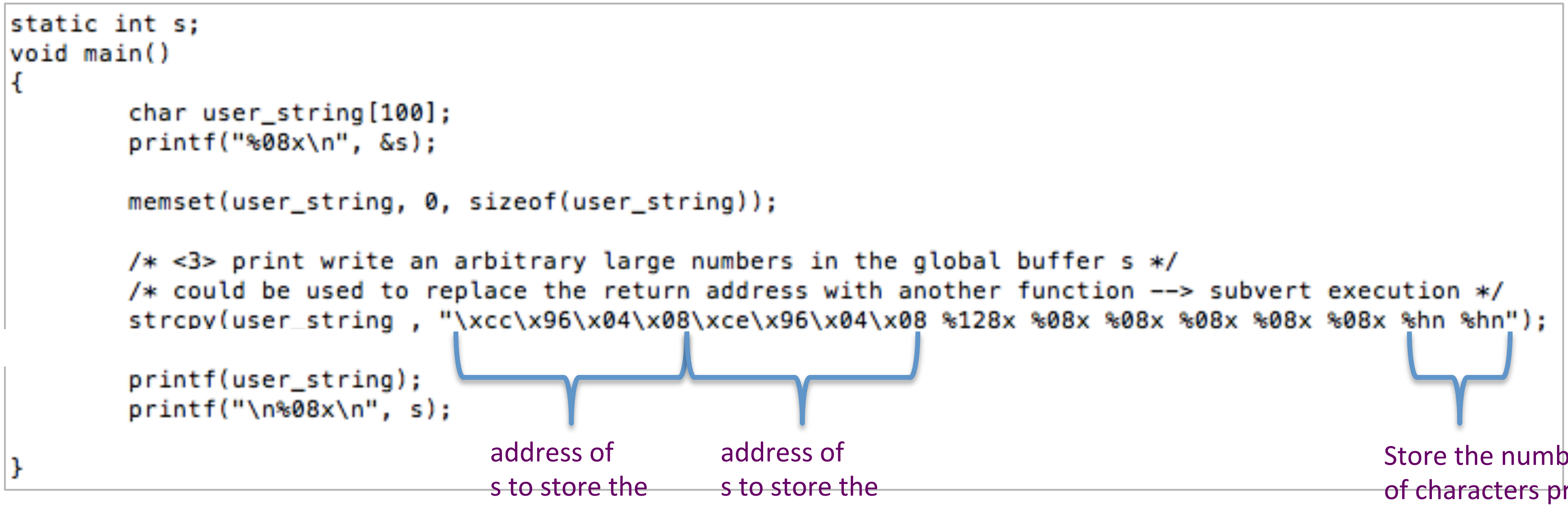


```
static int s;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */

        /* <2> write an arbitrary number in s */
        /* Change 50 to something else smaller and see the difference */
        strcpy(user_string , "\xa8\x96\x04\x08 %53x %7$n"); /* First 4 di
        printf(user_string);
        printf("\n%d\n", s);
}
```

An arbitrary number

Another useful format specifier

- %hn : will use only 16 bits .. Can be used to store large numbers



```
static int s;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));

        /* <3> print write an arbitrary large numbers in the global buffer s */
        /* could be used to replace the return address with another function --> subvert execution */
        strcpy(user_string , "\xcc\x96\x04\x08\xce\x96\x04\x08 %128x %08x %08x %08x %08x %08x %hn %hn");

        printf(user_string);
        printf("\n%08x\n", s);

}
```

address of s to store the lower 16bits

address of s to store the higher 16bits

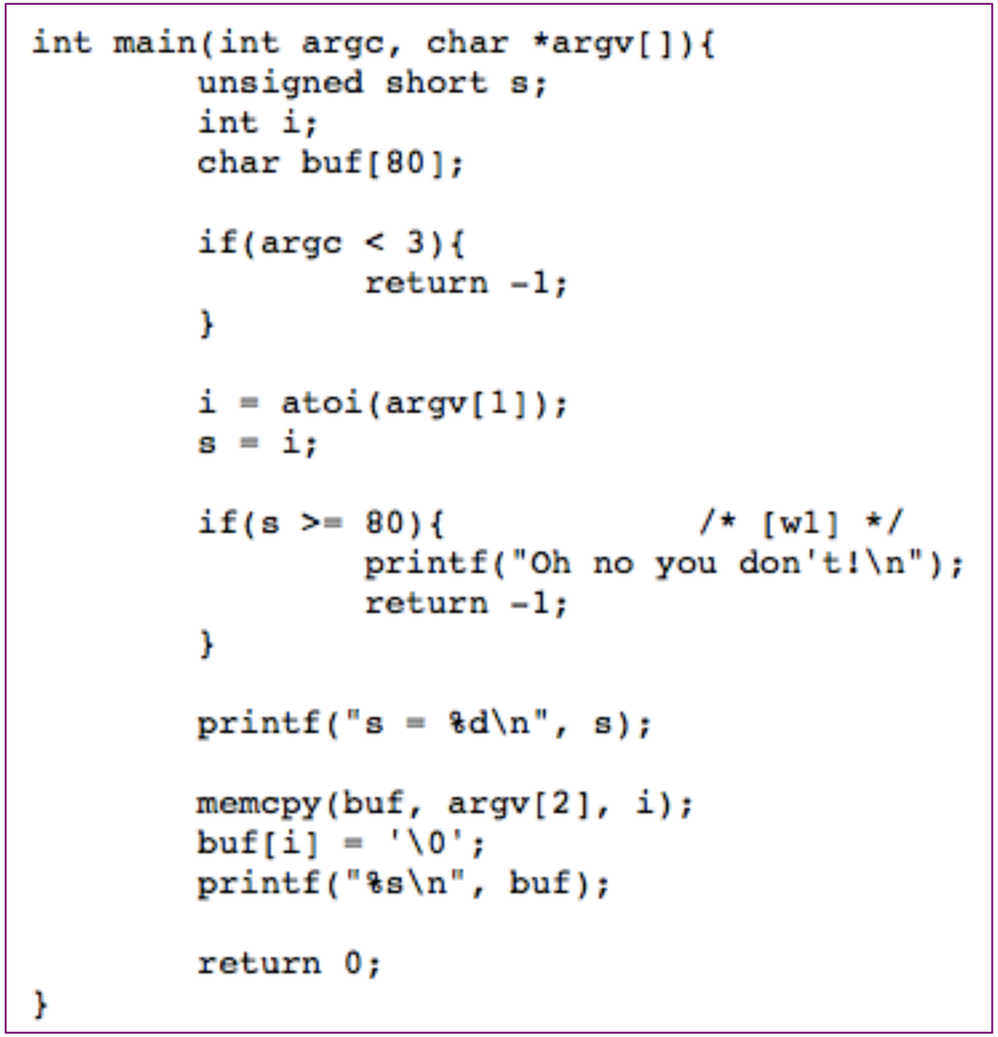
Store the number of characters printed.

Both 16 bit lower and 16 bit higher will be stored separately

Integer Overflow Vulnerability

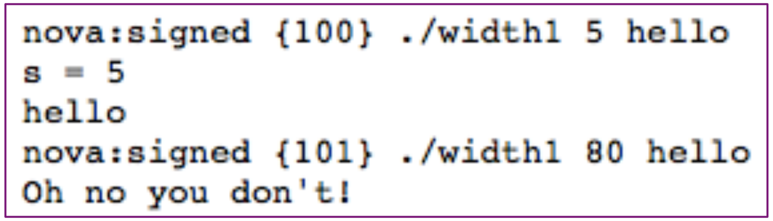
http://phrack.org/issues/60/10.html

What’s wrong with this code?



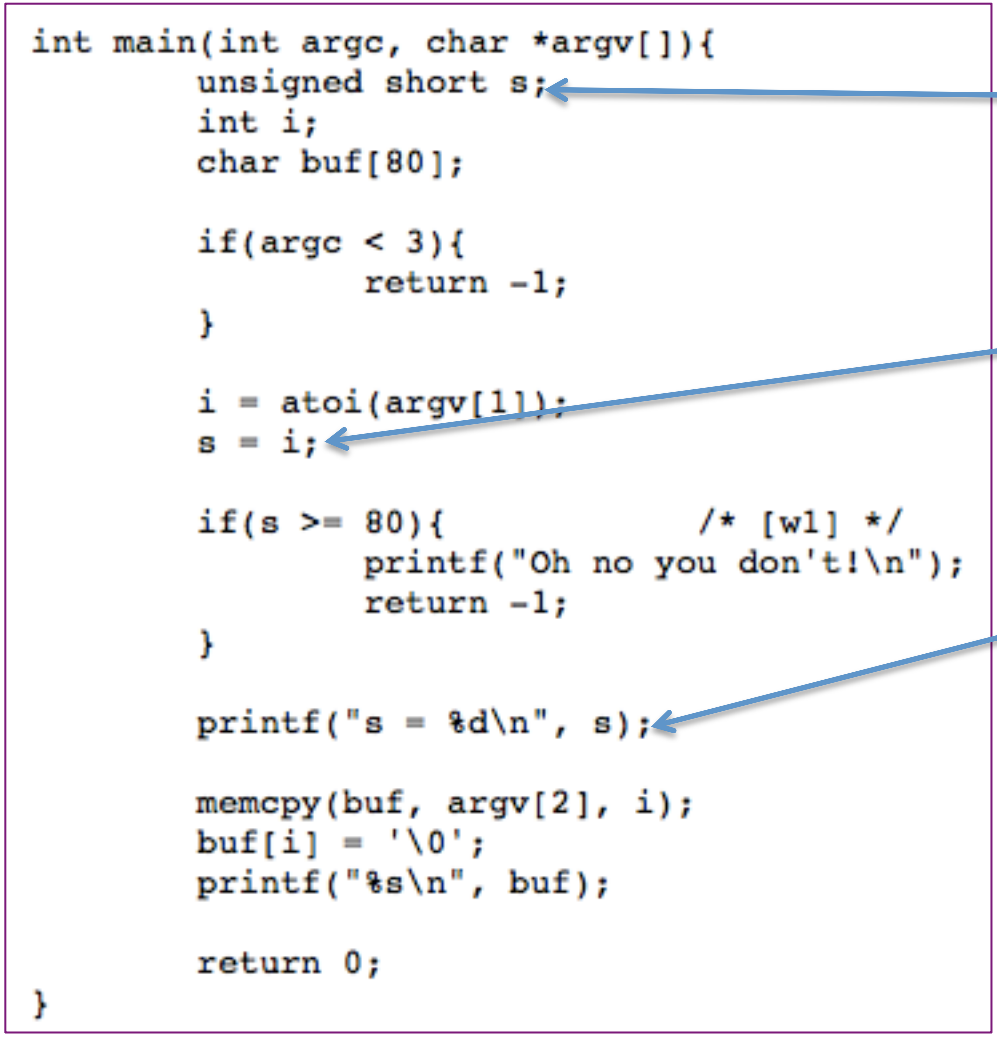
```c
int main(int argc, char *argv[]){
        unsigned short s;
        int i;
        char buf[80];

        if(argc < 3){
                return -1;
        }

        i = atoi(argv[1]);
        s = i;

        if(s >= 80){            /* [w1] */
                printf("Oh no you don't!\n");
                return -1;
        }

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

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

        return 0;
}
```

Expected behavior



```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
```

What’s wrong with this code?



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

        if(argc < 3){
                return -1;
        }

        i = atoi(argv[1]);
        s = i;

        if(s >= 80){            /* [w1] */
                printf("Oh no you don't!\n");
                return -1;
        }

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

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

        return 0;
}
```

Defined as short. Can hold a max value of 65535

If i > 65535, s overflows, therefore is truncated. So, the condition check is likely to be bypassed.

Will result in an overflow of buf, which can be used to perform nefarious activities

Integer Overflow Vulnerability

- Due to widthness overflow
- Due to arithmetic overflow
- Due to sign/unsigned problems

Widthness Overflows

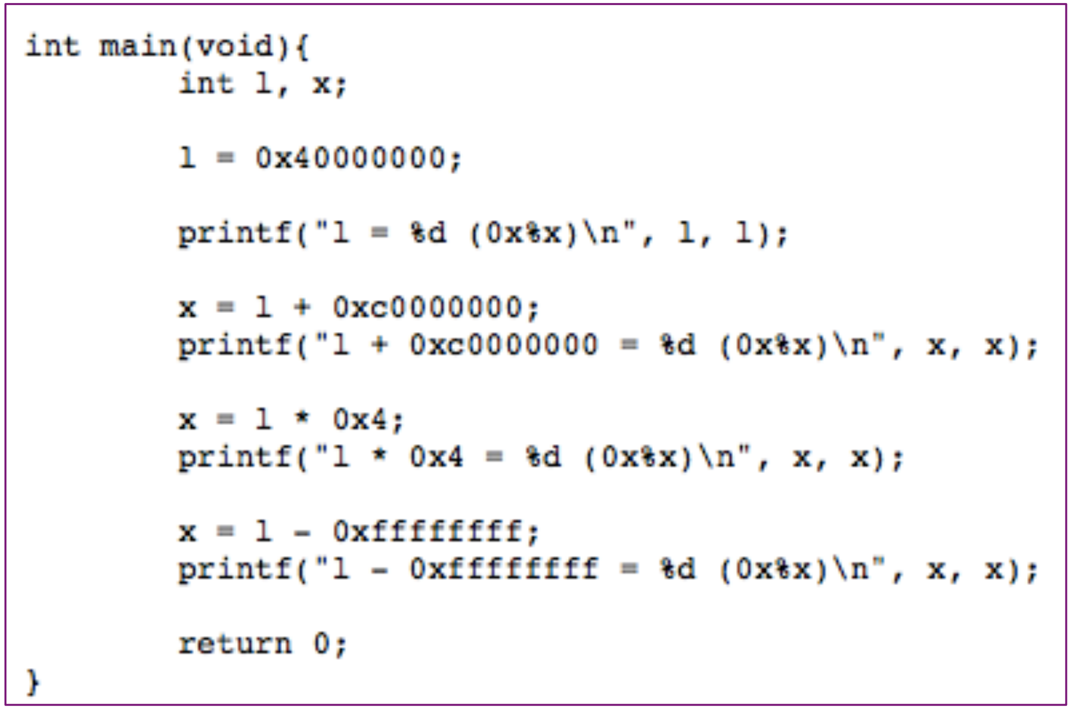
Occurs when code tries to store a value in a variable that is too small (in the number of bits) to handle it.

For example: a cast from int to short

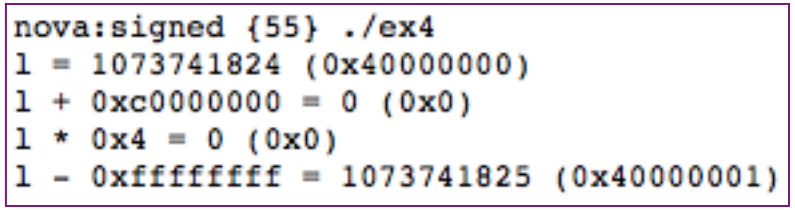
```
int a1 = 0x11223344;
char a2;
short a3;

a2 = (char) a1;
a3 = (short) a1;
```

```
a1 = 0x11223344
a2 = 0x44
a3 = 0x3344
```

Arithmetic Overflows

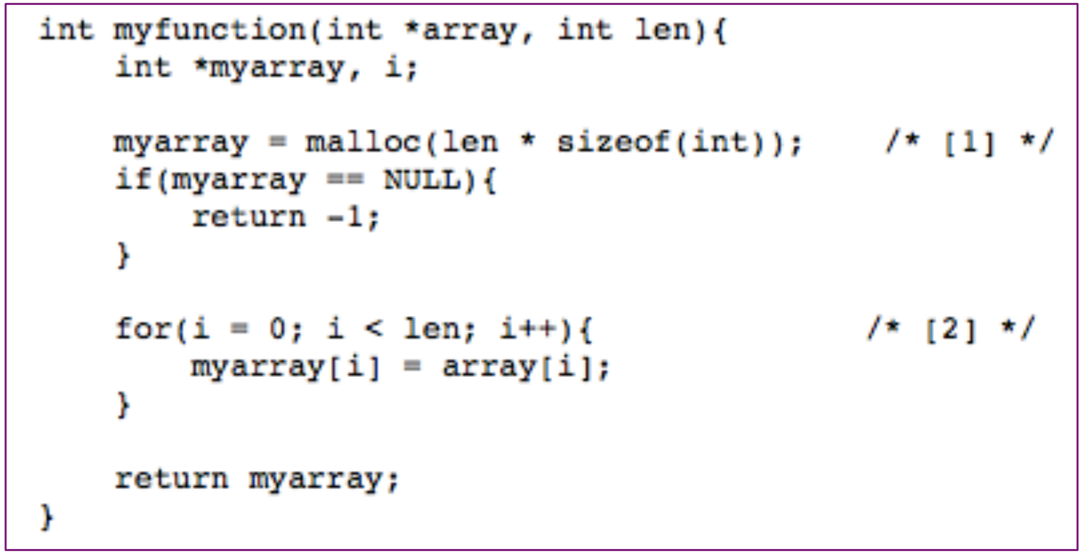


```
int main(void){
        int l, x;

        l = 0x40000000;

        printf("l = %d (0x%x)\n", l, l);

        x = l + 0xc0000000;
        printf("l + 0xc0000000 = %d (0x%x)\n", x, x);

        x = l * 0x4;
        printf("l * 0x4 = %d (0x%x)\n", x, x);

        x = l - 0xffffffff;
        printf("l - 0xffffffff = %d (0x%x)\n", x, x);

        return 0;
}
```



```
nova:signed {55} ./ex4
l = 1073741824 (0x40000000)
l + 0xc0000000 = 0 (0x0)
l * 0x4 = 0 (0x0)
l - 0xffffffff = 1073741825 (0x40000001)
```

Exploit 1
(manipulate space allocated by malloc)



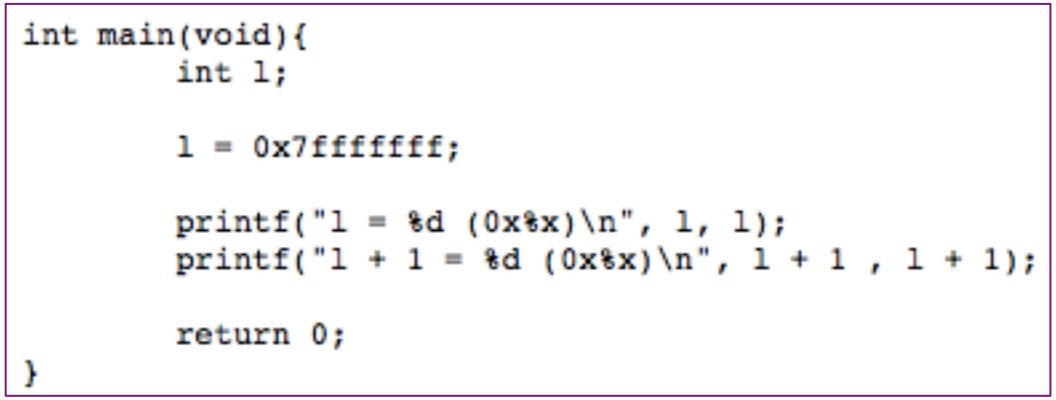
```
int myfunction(int *array, int len){
    int *myarray, i;

    myarray = malloc(len * sizeof(int));    /* [1] */
    if(myarray == NULL){
        return -1;
    }

    for(i = 0; i < len; i++){              /* [2] */
        myarray[i] = array[i];
    }

    return myarray;
}
```

Space allocated by malloc depends on len. If we choose a suitable value of len such that len*sizeof(int) overflows, then,

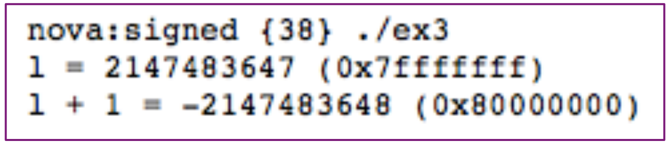
(1) myarray would be smaller than expected
(2) thus leading to a heap overflow
(3) which can be exploited

(Un)signed Integers

- Sign interpreted using the most significant bit.
- This can lead to unexpected results in comparisons and arithmetic



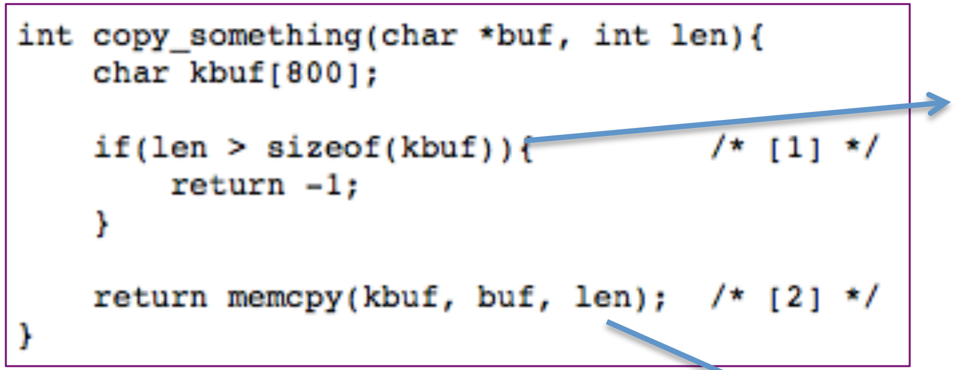
```
int main(void){
        int l;

        l = 0x7fffffff;

        printf("l = %d (0x%x)\n", l, l);
        printf("l + 1 = %d (0x%x)\n", l + 1 , l + 1);

        return 0;
}
```



```
nova:signed {38} ./ex3
l = 2147483647 (0x7fffffff)
l + 1 = -2147483648 (0x80000000)
```

i is initialized with the highest positive value that a signed 32 bit integer can take.
When incremented, the MSB is set, and the number is interpreted as negative.

Sign Interpretations in compare



```
int copy_something(char *buf, int len){
    char kbuf[800];

    if(len > sizeof(kbuf)){          /* [1] */
        return -1;
    }

    return memcpy(kbuf, buf, len);   /* [2] */
}
```

This test is with signed numbers. Therefore a negative len will pass the ‘if’ test.

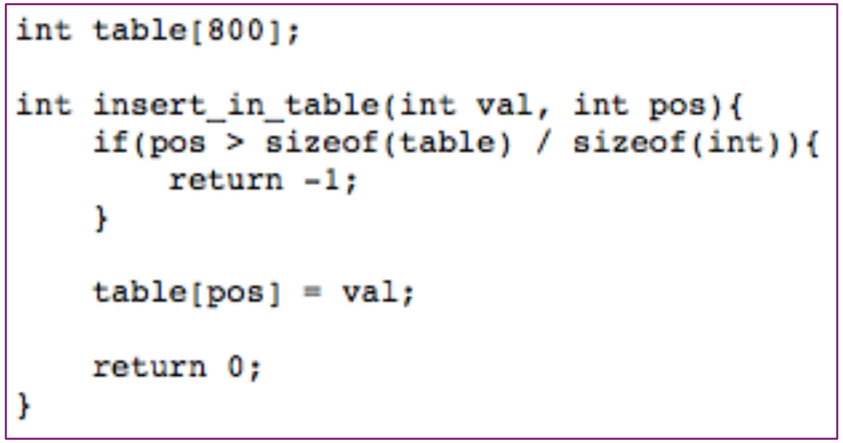
In memcpy, len is interpreted as unsigned. Therefore a negative len will be treated as positive.

This could be used to overflow kbuf.

From the man pages

void ***memcpy**(void *restrict dst, const void *restrict src, size_t n);

Sign interpretations in arithmetic

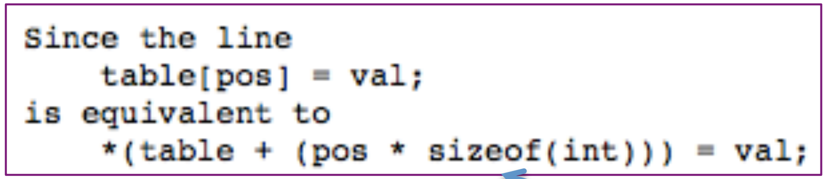


```
int table[800];

int insert_in_table(int val, int pos){
    if(pos > sizeof(table) / sizeof(int)){
        return -1;
    }

    table[pos] = val;

    return 0;
}
```

table + pos is expected to be a value greater than table.

If *pos* is negative, this is not the case.

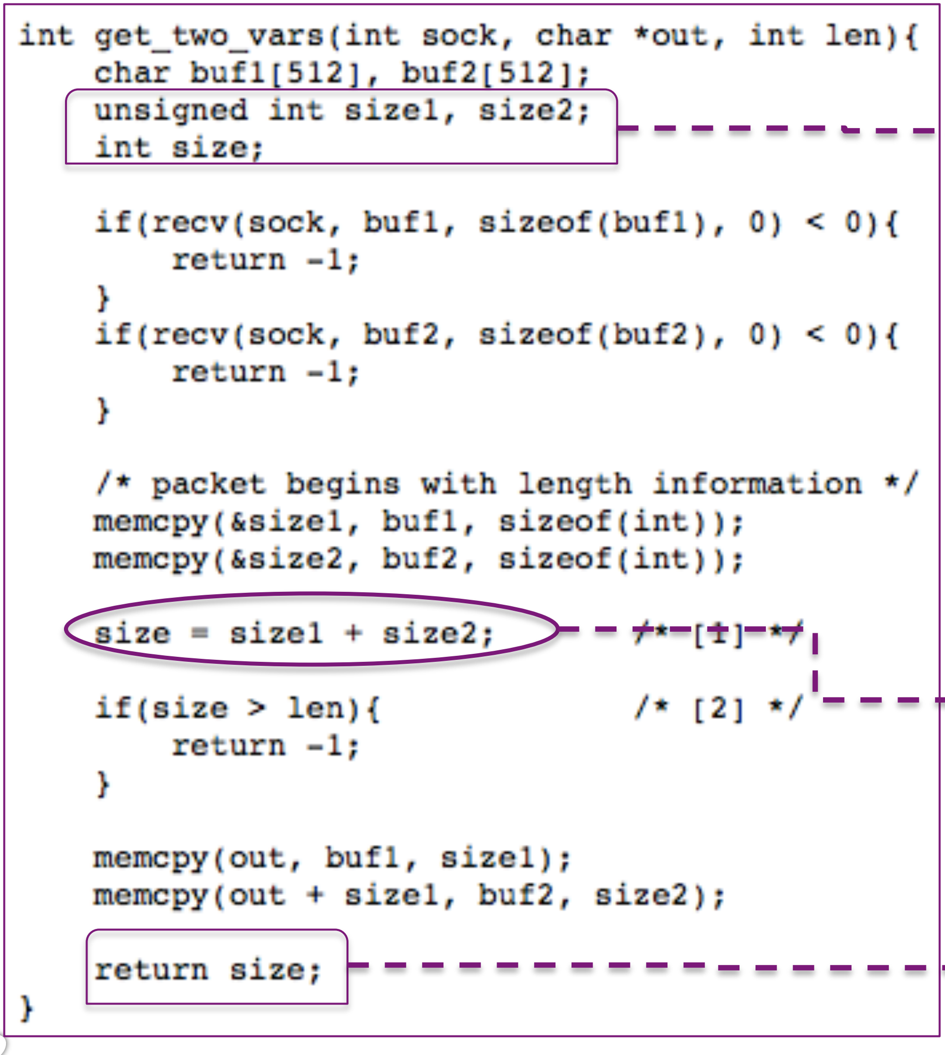
Causing *val* to be written to a location beyond the table



```
Since the line
    table[pos] = val;
is equivalent to
    *(table + (pos * sizeof(int))) = val;
```

This arithmetic done considering unsigned

exploiting overflow due to sign in a network deamon



```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;          /* [1] */

    if(size > len){                /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}
```

size1 and size2 are unsigned
Size is signed.

if size1 and size2 are large enough,
size may end up being negative.

Size is returned, which may
cause an out to overflow in the
callee function

Sign could lead to memory overreads.

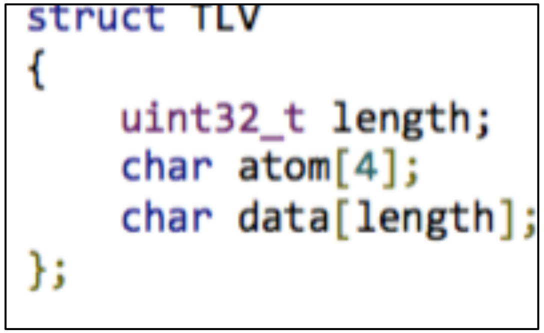
```
#define MAX_BUF_SIZE 64 * 1024

void store_into_buffer(const void *src, int num)
{
  char global_buffer[MAX_BUF_SIZE];

  if (num > MAX_BUF_SIZE)
    return;

  memcpy(global_buffer, src, num);

  [...]
}
```

- num is a signed int
- If num is negative, then it will pass the if test
- memcpy’s 3rd parameter is unsigned. So, the negative number is interpreted as positive. Resulting in memory overreads.

Stagefright Bug

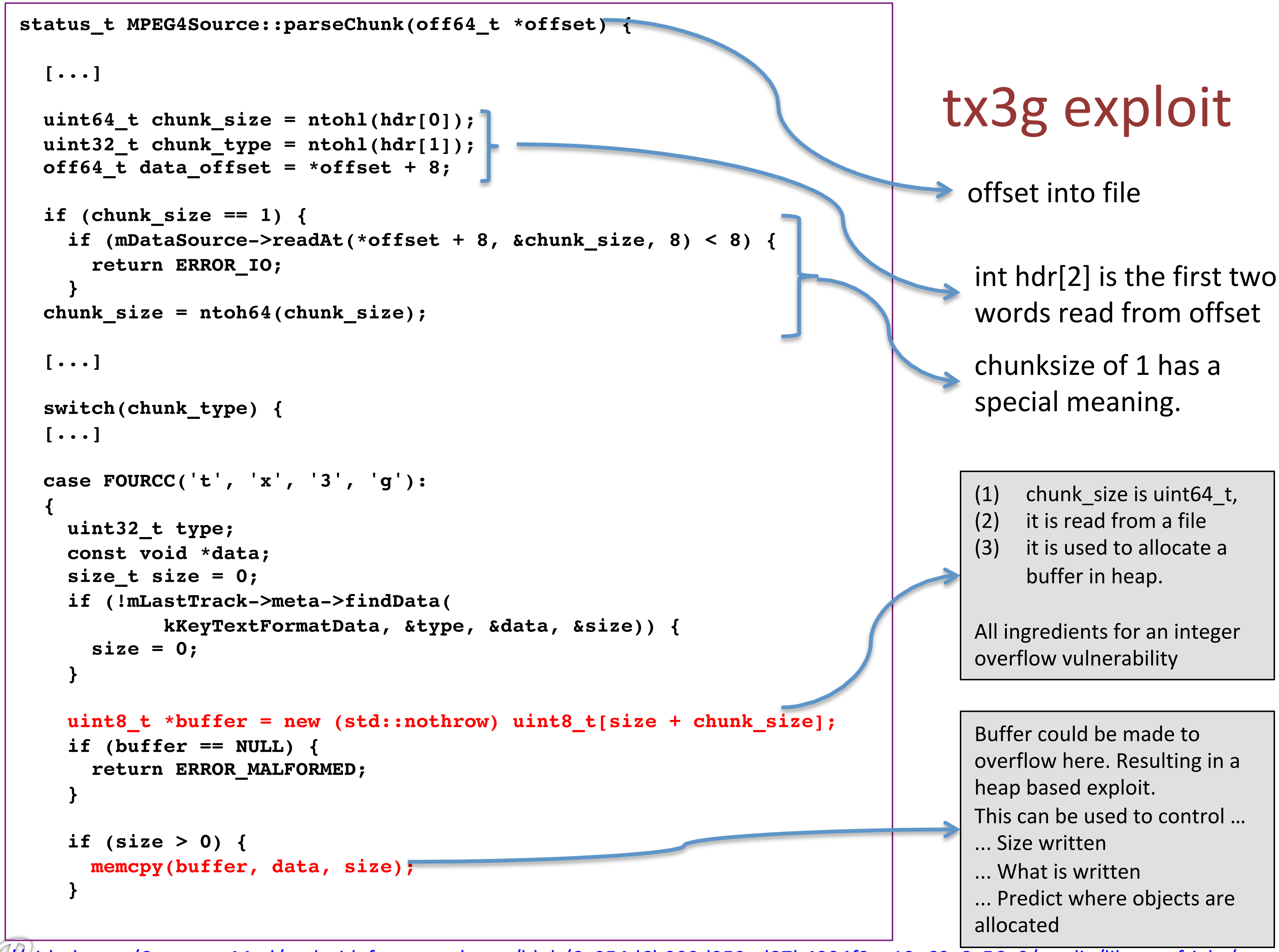
- Discovered by Joshua Drake and disclosed on July 27th, 2015
- Stagefright is a software library implemented in C++ for Android
- Stagefright attacks uses several integer based bugs to
  - execute remote code in phone
  - Achieve privilige escalation
- Attack is based on a well crafted MP3, MP4 message sent to the remote Android phone
  - Multiple vulnerabilities exploited:
    - One exploit targets MP4 subtitles that uses tx3g for timed text.
    - Another exploit targets covr (cover art) box
- Could have affected around one thousand million devices
  - Devices affected inspite of ASLR

MPEG4 Format



```
struct TLV
{
    uint32_t length;
    char atom[4];
    char data[length];
};
```

tx3g exploit



```
status_t MPEG4Source::parseChunk(off64_t *offset) {

  [...]

  uint64_t chunk_size = ntohl(hdr[0]);
  uint32_t chunk_type = ntohl(hdr[1]);
  off64_t data_offset = *offset + 8;

  if (chunk_size == 1) {
    if (mDataSource->readAt(*offset + 8, &chunk_size, 8) < 8) {
      return ERROR_IO;
    }
  chunk_size = ntoh64(chunk_size);

  [...]

  switch(chunk_type) {
  [...]

  case FOURCC('t', 'x', '3', 'g'):
  {
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
      size = 0;
    }

    uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
    if (buffer == NULL) {
      return ERROR_MALFORMED;
    }

    if (size > 0) {
      memcpy(buffer, data, size);
    }
```

offset into file

int hdr[2] is the first two words read from offset

chunksize of 1 has a special meaning.

(1) chunk_size is uint64_t,
(2) it is read from a file
(3) it is used to allocate a buffer in heap.

All ingredients for an integer overflow vulnerability

Buffer could be made to overflow here. Resulting in a heap based exploit.
This can be used to control …
… Size written
… What is written
… Predict where objects are allocated

https://github.com/CyanogenMod/android_frameworks_av/blob/6a054d6b999d252ed87b4224f3aa13e69e3c56e0/media/libstagefright/MPEG4Extractor.cpp#L1954

Integer Overflows

```
uint64_t chunk_size = ntohl(hdr[0]);

uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
```

On 32 bit platforms

widthness overflow

(chunk_size + size) is uint64_t however new takes a 32 bit value

On 64 bit platforms

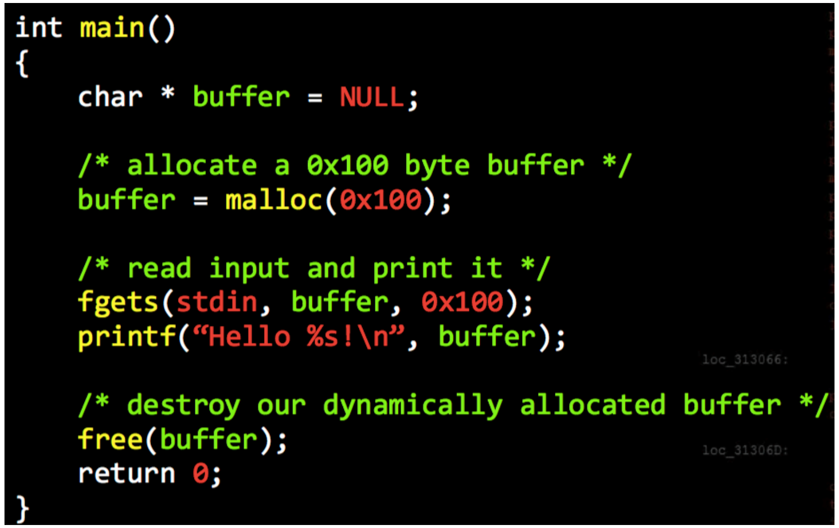
arithmetic overflow

(chunk_size + size) can overflow by setting large values for chunk_size

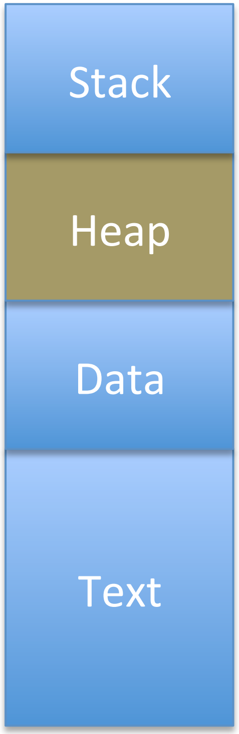
Heap exploits

Heap

- Just a pool of memory used for dynamic memory allocation



```c
int main()
{
    char * buffer = NULL;

    /* allocate a 0x100 byte buffer */
    buffer = malloc(0x100);

    /* read input and print it */
    fgets(stdin, buffer, 0x100);
    printf(“Hello %s!\n”, buffer);

    /* destroy our dynamically allocated buffer */
    free(buffer);
    return 0;
}
```



Heap vs Stack

- Heap
  - Slow
  - Manually done by free and malloc
  - Used for objects, large arrays, persistent data (across function calls)

- Stack
  - Fast
  - Automatically done by compiler
  - Temporary data store

Heap Management

- Several different types of implementations
  - Doug Lea’s forms the base for many
  - glibc uses ptmalloc
  - Others include

    tcmalloc

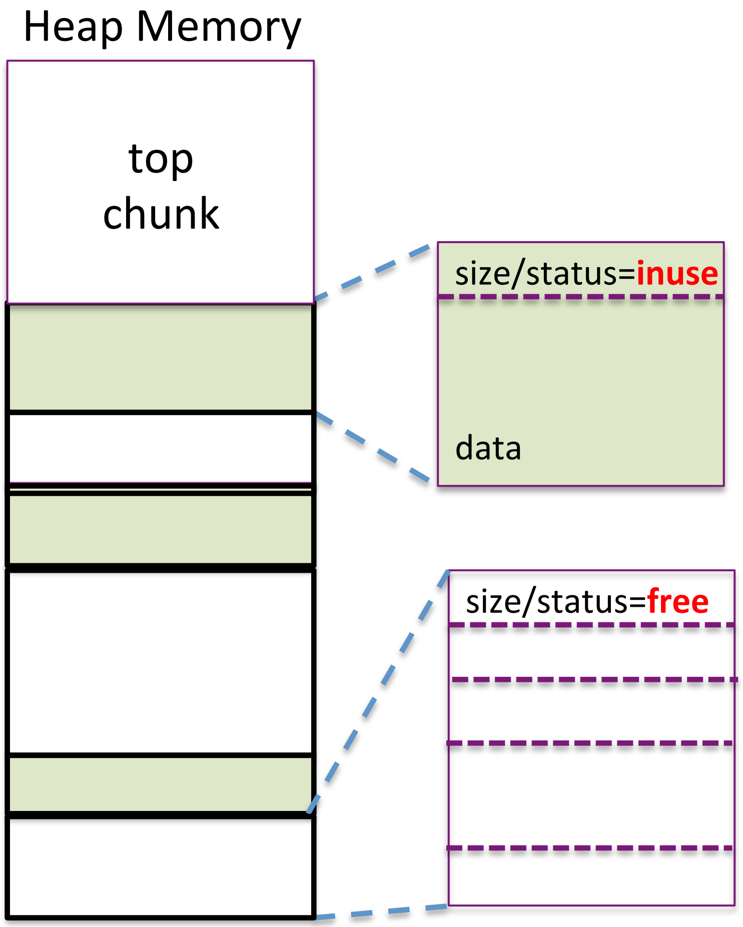
    jemalloc (used in Android)

    nedmalloc

    Hoard

http://gee.cs.oswego.edu
ftp://g.oswego.edu/pub/misc/malloc.c
ptmalloc

Doug Lea’s Malloc



Heap Memory split into chunks of various sizes

Free chucks :
Two bordering unused chunks can be coalesced into one larger chunk

All free chunks can be traversed via linked lists (double or single)

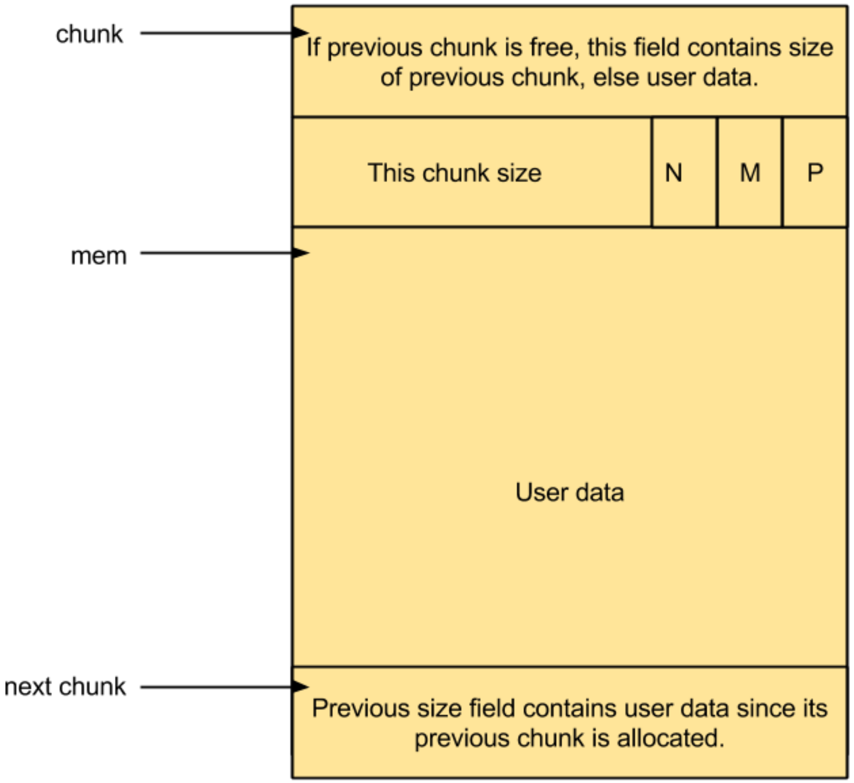
If correct sized chunk is unavailable, a larger chunk can be split

Allocated chunks:
To find the next used chunk compute size + base_address
All allocated chunks either border a free chunk or the top chunk

http://g.oswego.edu/dl/html/malloc.html

glib's structures

Allocated chunk


chunk
If previous chunk is free, this field contains size of previous chunk, else user data.
This chunk size
N
M
P
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.

Allocated Chunk

P : previous chunk in use (PREV_INUSE bit)

If P=0, then the word before this contains the size of the previous chunk.

The very first chunk always has this bit set
Preventing access to non-existent memory.

M : set if chunk was obtained with mmap

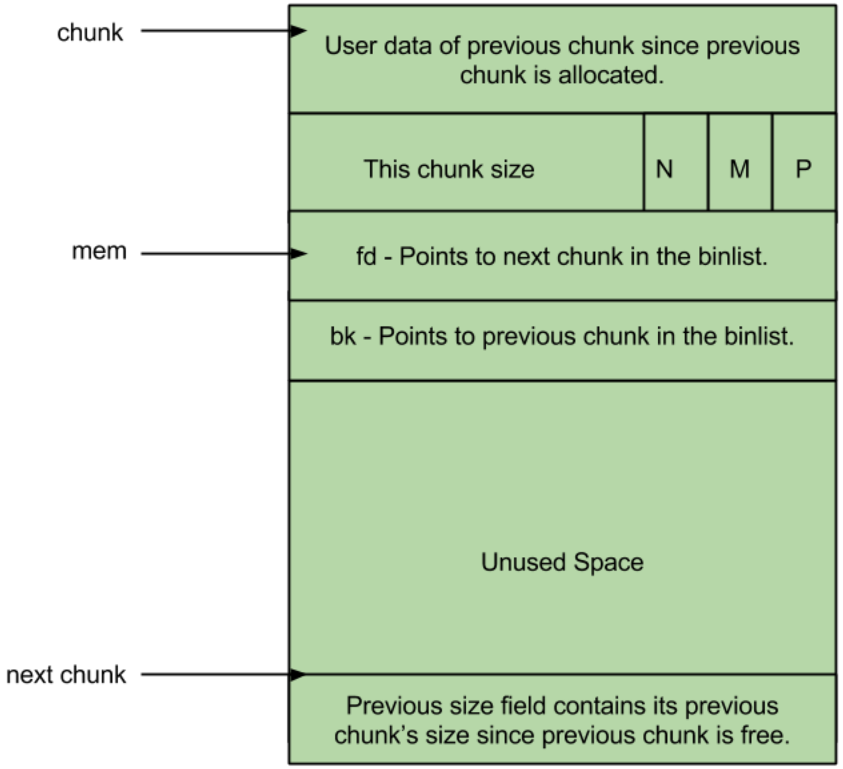
A : set if chunk belongs to thread arena

mem. Is the pointer returned by malloc.
chunk. Is the pointer to metadata for malloc

User data size for malloc(n) is
N = 8 + (n/8)*8 bytes.
Total size of chunk is N+8 bytes

glib’s structures

Free chunk



chunk

User data of previous chunk since previous chunk is allocated.

This chunk size | N | M | P

mem

fd - Points to next chunk in the binlist.

bk - Points to previous chunk in the binlist.

Unused Space

next chunk

Previous size field contains its previous chunk’s size since previous chunk is free.

Free Chunk

P : previous chunk in use (PREV_INUSE bit)

If P=0, then the word before this contains the size of the previous chunk.

The very first chunk always has this bit set
Preventing access to non-existent memory.

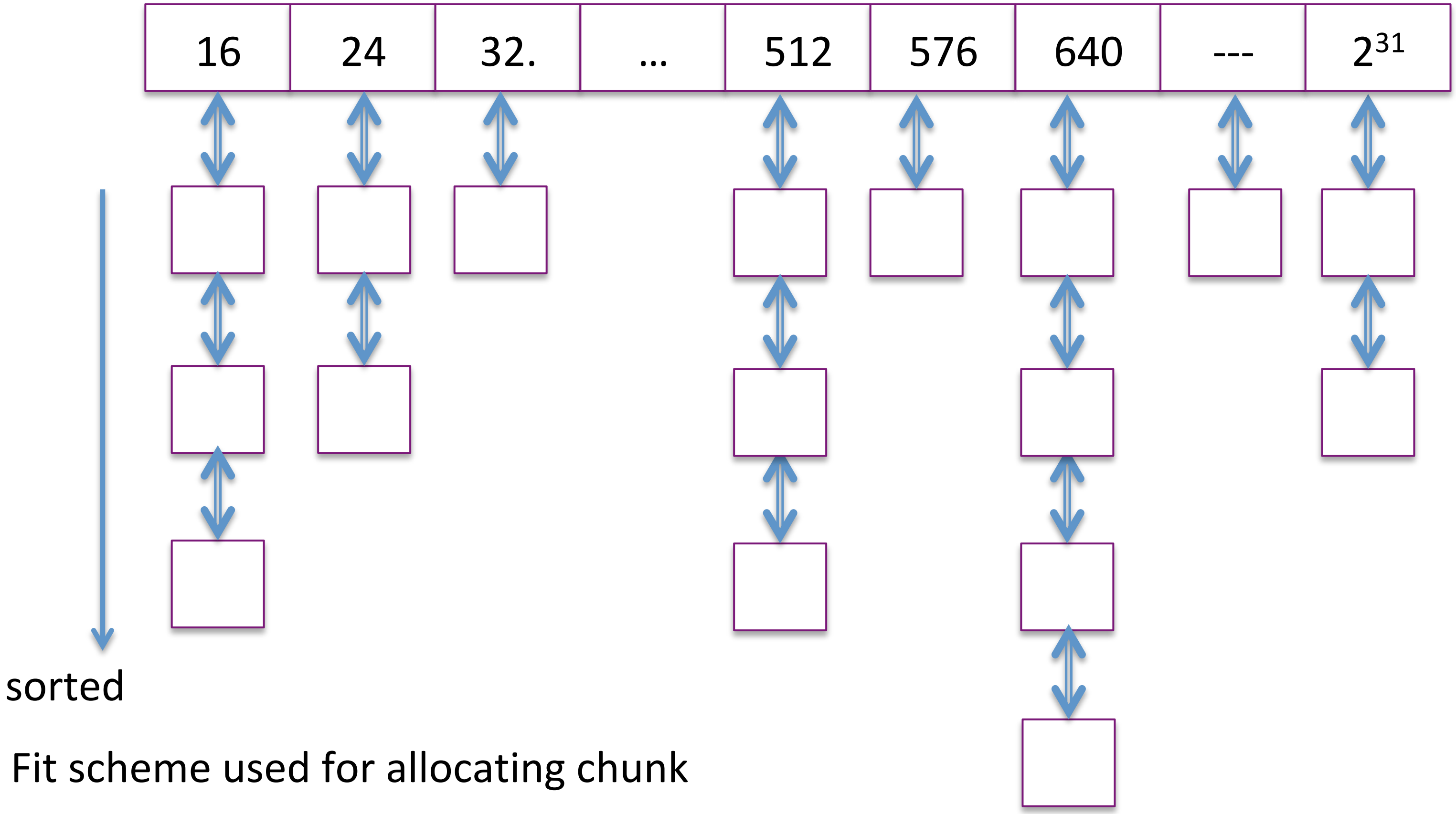
M : set if chunk was obtained with mmap

A : set if chunk belongs to thread arena

mem. Is the pointer returned by malloc.
chunk. Is the pointer to metadata for malloc

User data size for malloc(n) is
N = 8 + (n/8)*8 bytes.
Total size of chunk is N+8 bytes

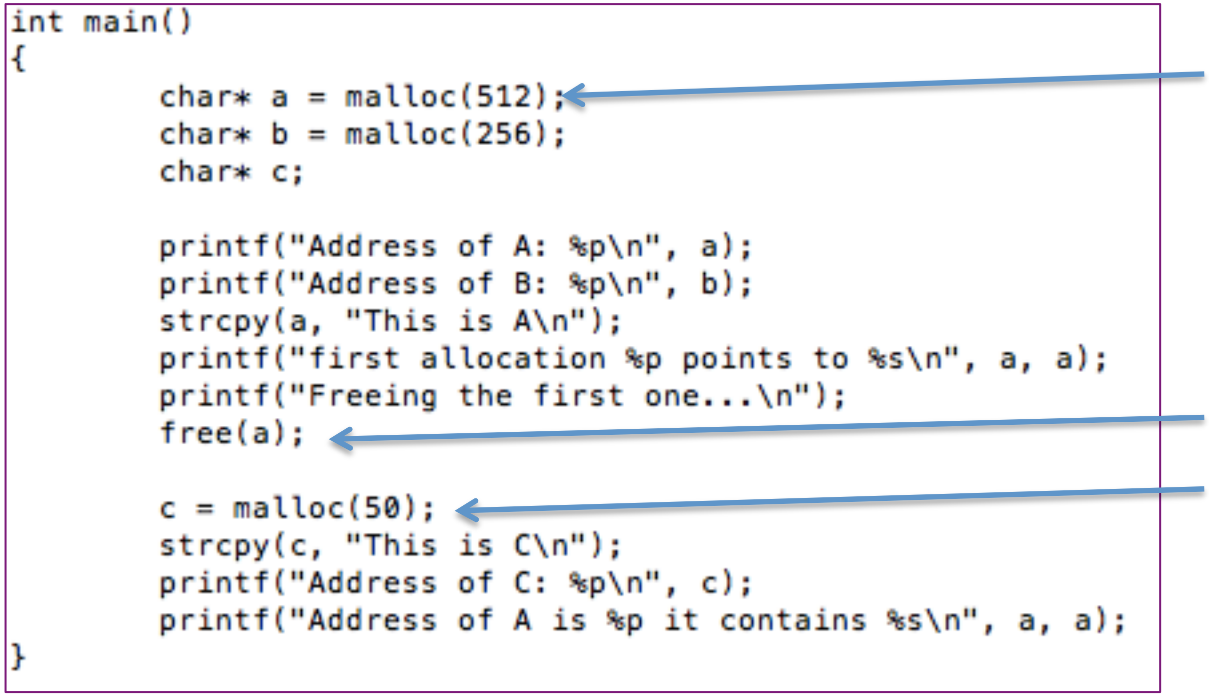
Binning



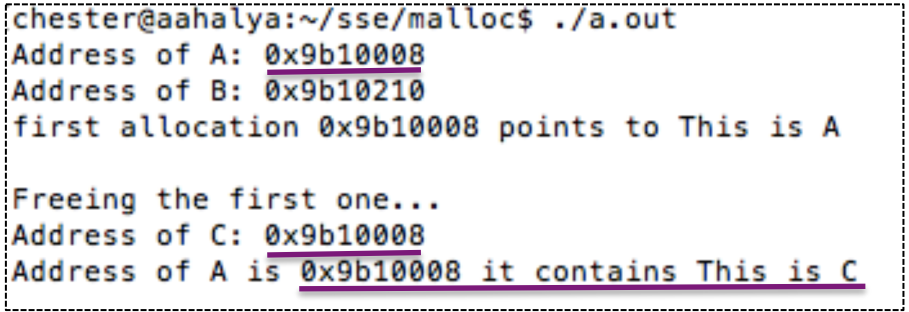
First Fit scheme used for allocating chunk

Glib’s first fit allocator

First Fit scheme used for allocating chunk



```
int main()
{
        char* a = malloc(512);
        char* b = malloc(256);
        char* c;

        printf("Address of A: %p\n", a);
        printf("Address of B: %p\n", b);
        strcpy(a, "This is A\n");
        printf("first allocation %p points to %s\n", a, a);
        printf("Freeing the first one...\n");
        free(a);

        c = malloc(50);
        strcpy(c, "This is C\n");
        printf("Address of C: %p\n", c);
        printf("Address of A is %p it contains %s\n", a, a);
}
```



```
chester@aahalya:~/sse/malloc$ ./a.out
Address of A: 0x9b10008
Address of B: 0x9b10210
first allocation 0x9b10008 points to This is A

Freeing the first one...
Address of C: 0x9b10008
Address of A is 0x9b10008 it contains This is C
```

Allocating a memory chunk of 512 bytes

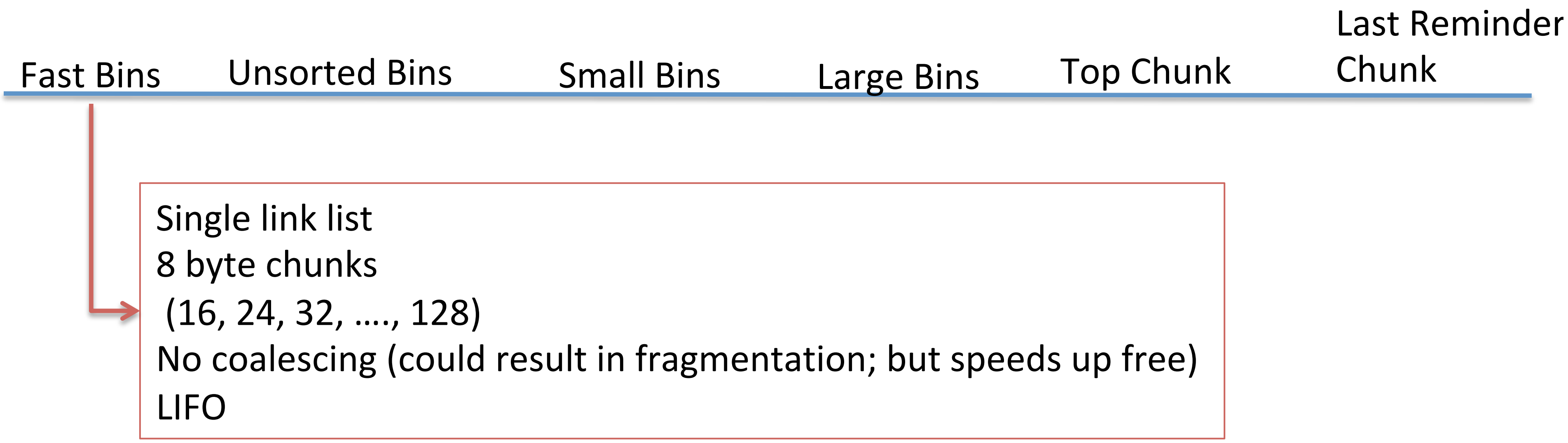
Now freeing it

Now allocating another chunk < 512 bytes.

The first free chunk available corresponds to the freed ‘a’. So, ‘c’ gets allocated the same address as ‘a’

https://github.com/shellphish/how2heap (first_fit.c)

Types of Bins

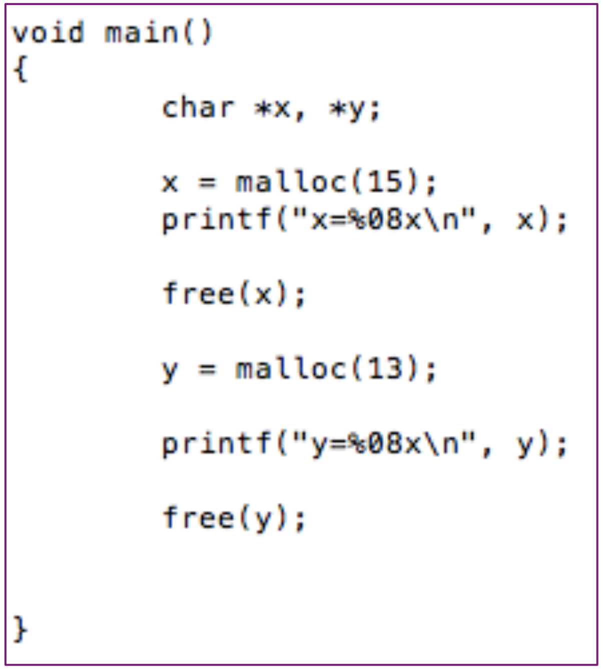


Fast Bins Unsorted Bins Small Bins Large Bins Top Chunk Last Reminder Chunk

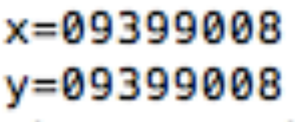
Single link list
8 byte chunks
(16, 24, 32, …., 128)
No coalescing (could result in fragmentation; but speeds up free)
LIFO

Example of Fast Binning

x and y end up in the same bin.

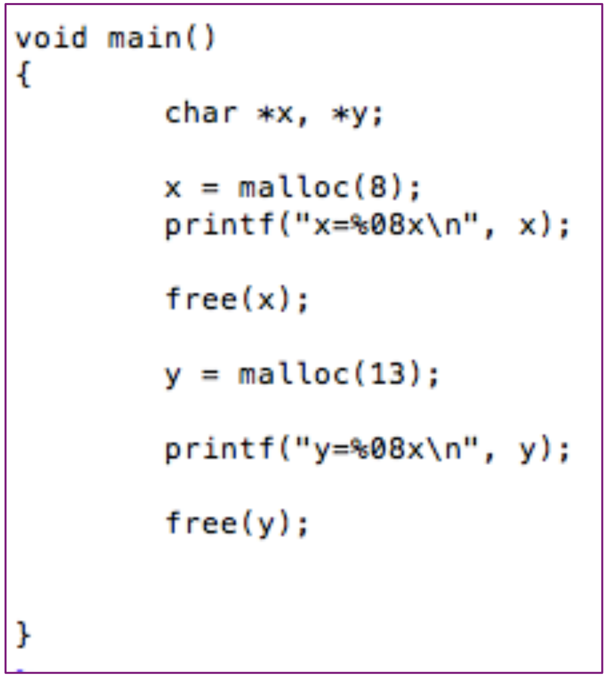


```
void main()
{
        char *x, *y;

        x = malloc(15);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);


}
```

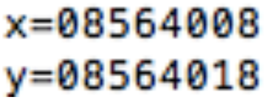


```
x=09399008
y=09399008
```

x and y end up in different bins.



```
void main()
{
        char *x, *y;

        x = malloc(8);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);


}
```



```
x=08564008
y=08564018
```

Types of Bins

Fast Bins **Unsorted Bins** Small Bins Large Bins Top Chunk Last Reminder Chunk

Single link list

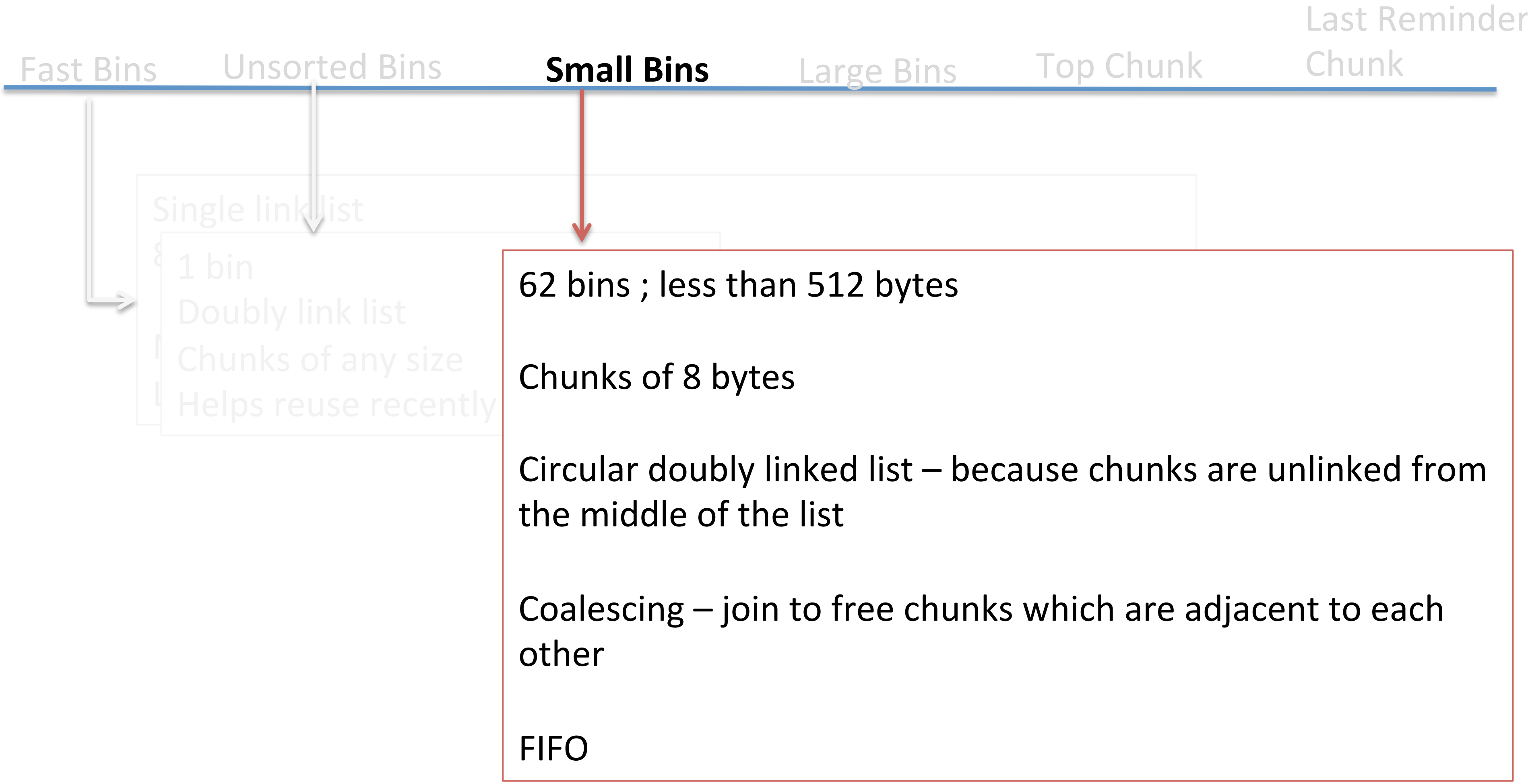
8

entation; but speeds up free)

1 bin
Doubly link list
Chunks of any size
Helps reuse recently used chunks

Uses the first chunk that fits.

Types of Bins



Fast Bins

Unsorted Bins

Small Bins

Large Bins

Top Chunk

Last Reminder Chunk

Single link list

1 bin

Doubly link list

Chunks of any size

Helps reuse recently

62 bins ; less than 512 bytes

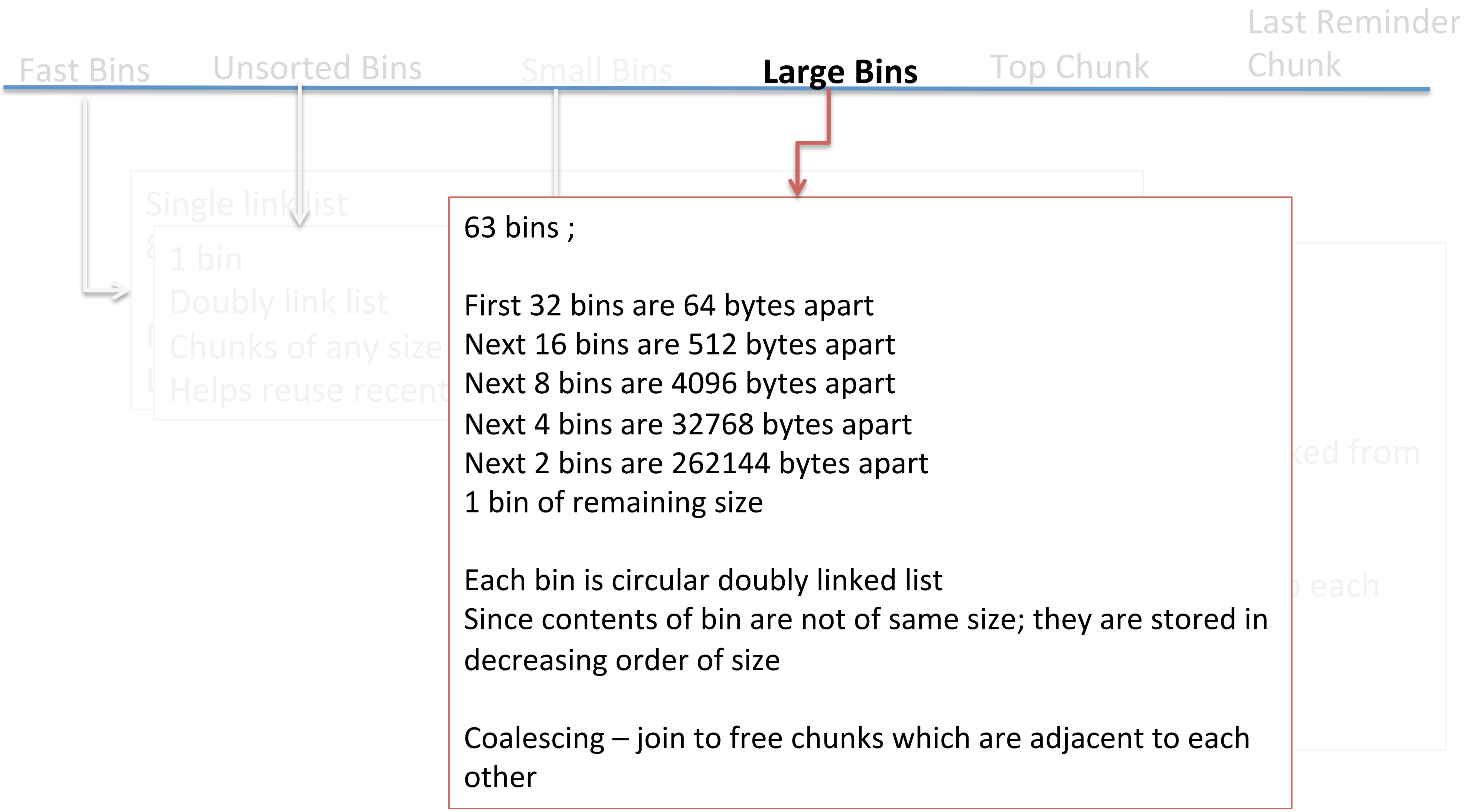
Chunks of 8 bytes

Circular doubly linked list – because chunks are unlinked from the middle of the list

Coalescing – join to free chunks which are adjacent to each other

FIFO

Types of Bins



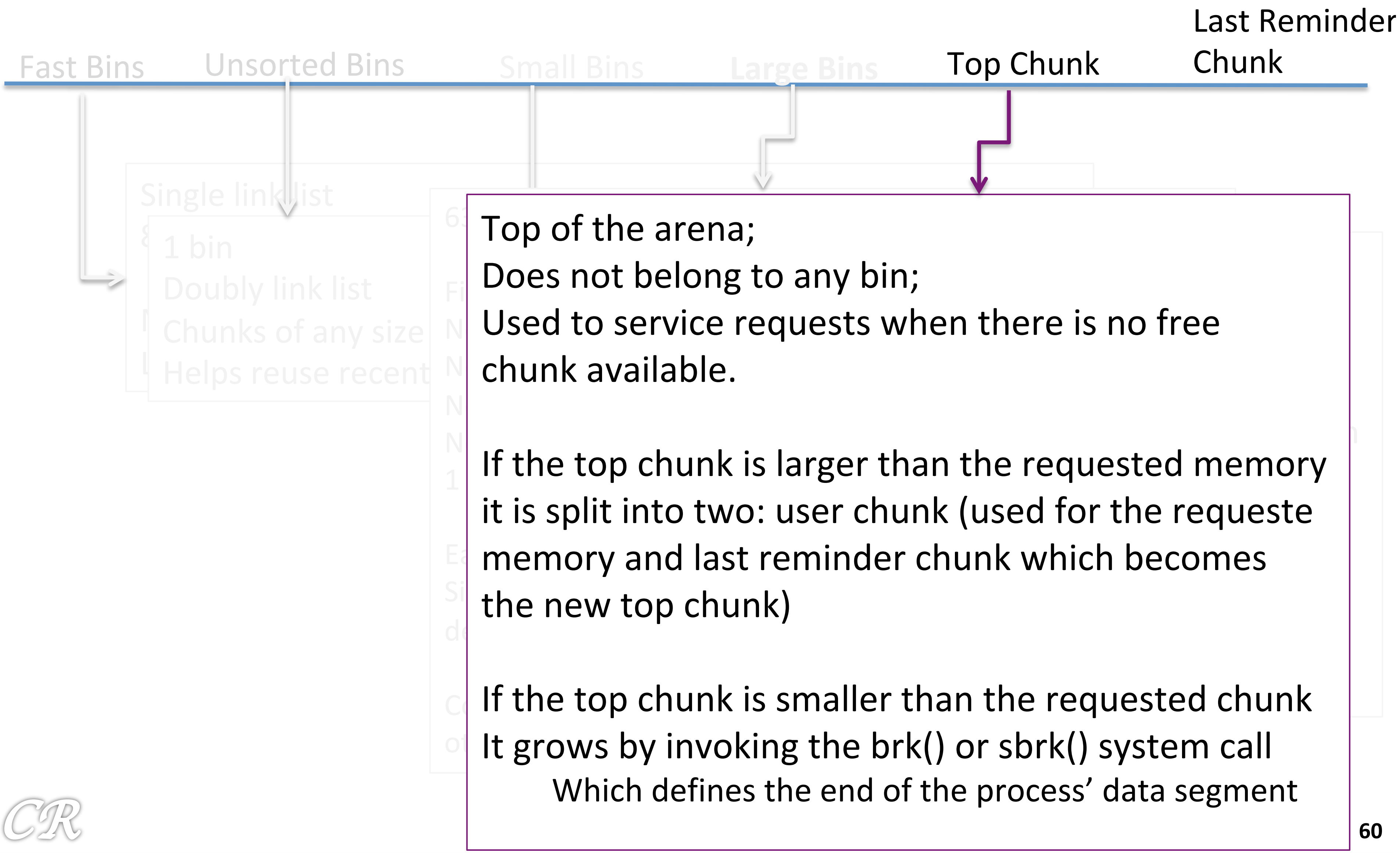
63 bins ;

First 32 bins are 64 bytes apart
Next 16 bins are 512 bytes apart
Next 8 bins are 4096 bytes apart
Next 4 bins are 32768 bytes apart
Next 2 bins are 262144 bytes apart
1 bin of remaining size

Each bin is circular doubly linked list
Since contents of bin are not of same size; they are stored in decreasing order of size

Coalescing – join to free chunks which are adjacent to each other

Types of Bins



Top of the arena;
Does not belong to any bin;
Used to service requests when there is no free chunk available.

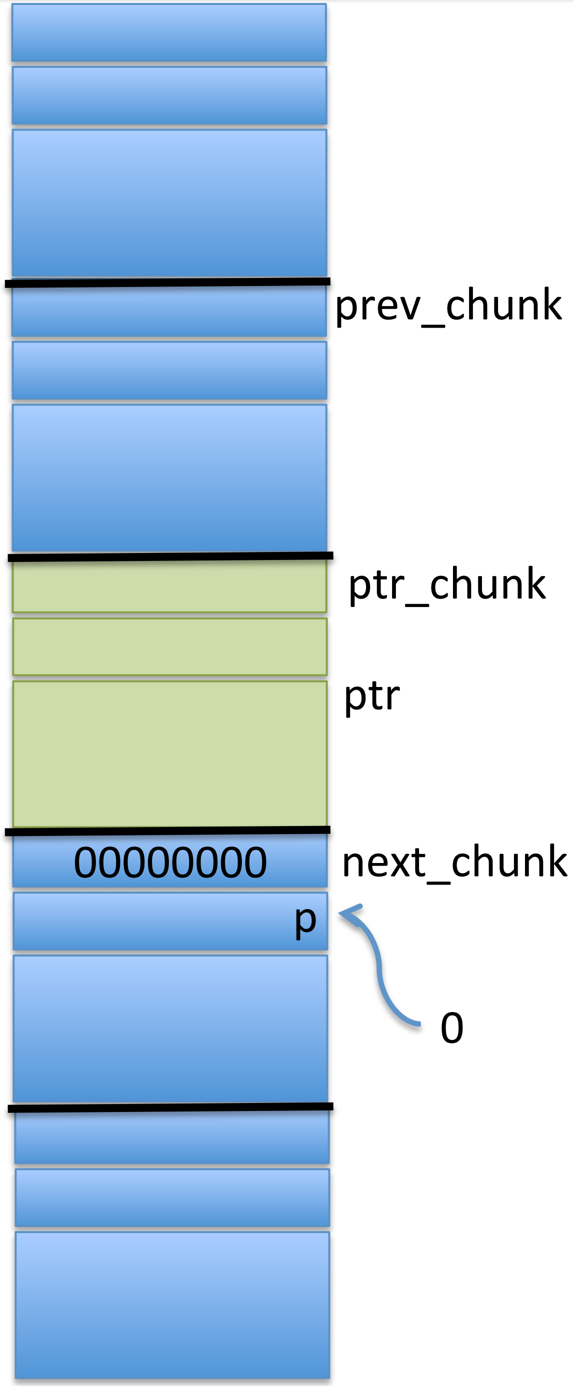
If the top chunk is larger than the requested memory it is split into two: user chunk (used for the requeste memory and last reminder chunk which becomes the new top chunk)

If the top chunk is smaller than the requested chunk
It grows by invoking the brk() or sbrk() system call

Which defines the end of the process' data segment

free(ptr)

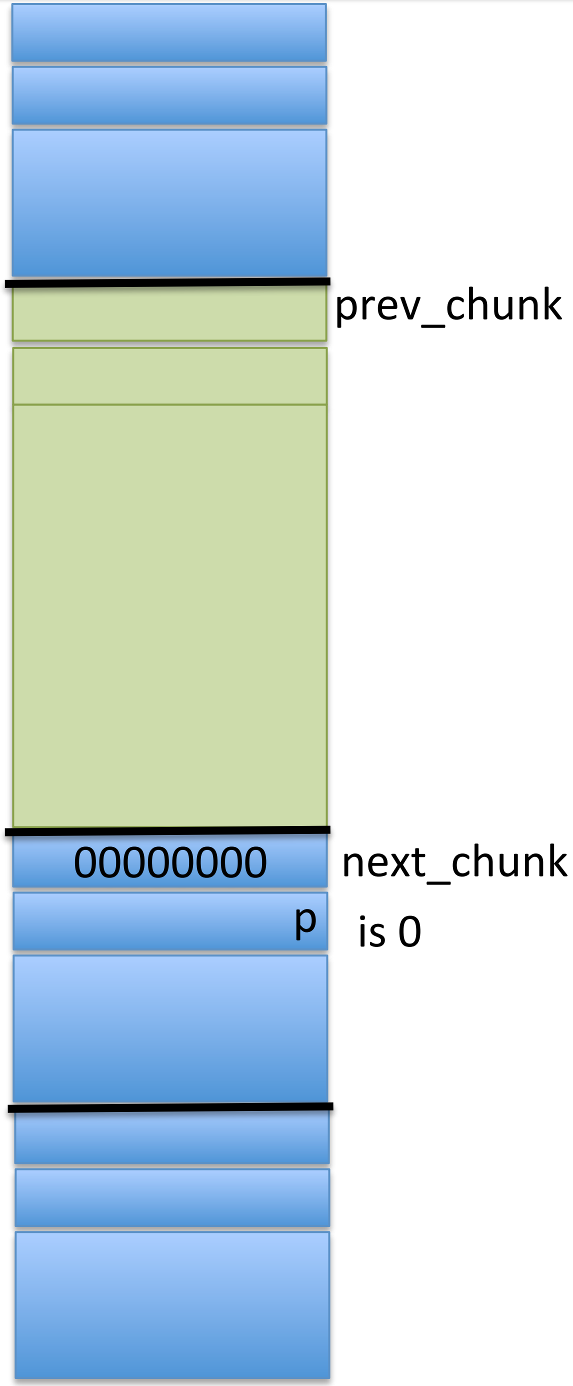
1. If the next chunk is allocated then
   - Set size to zero
   - Set p bit to 0



free(ptr)

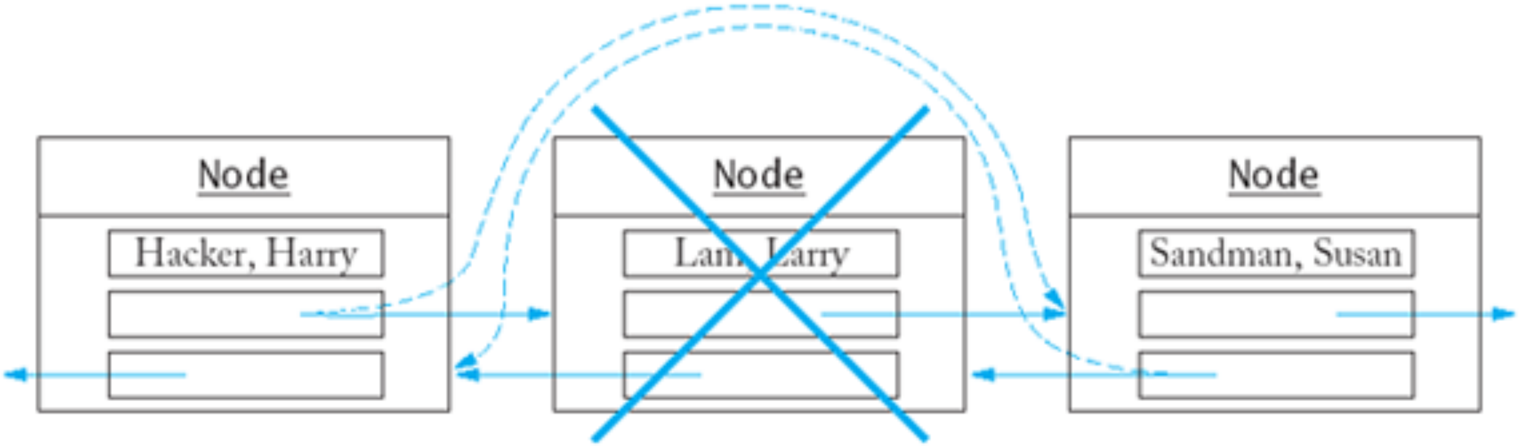
2. If the previous chunk is free then
   - Coalesce the two to create a new free chunk
   - This will also require unlinking from the current bin and placing the larger chunk in the appropriate bin

Similar is done if the next chuck is free as well.



Unlinking from a free list

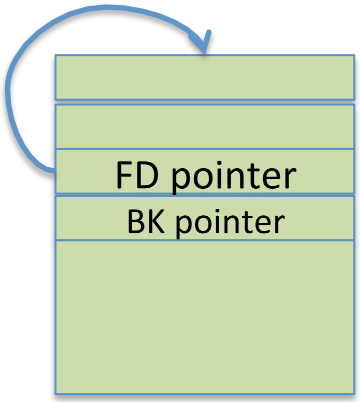
```
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD){
     FD = P->fd;
     BK = P->bk;
     FD->bk = BK;
     BK->fd = FD;
}
```



More recent Unlinking

```
/* Take a chunk off a bin list */
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD)
{
      FD = P->fd;
      BK = P->bk;
      if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
            malloc_printerr(check_action,"corrupted double-linked list",P);
      else {
            FD->bk = BK;
            BK->fd = FD;
      }
}
```

Detects cases such as these

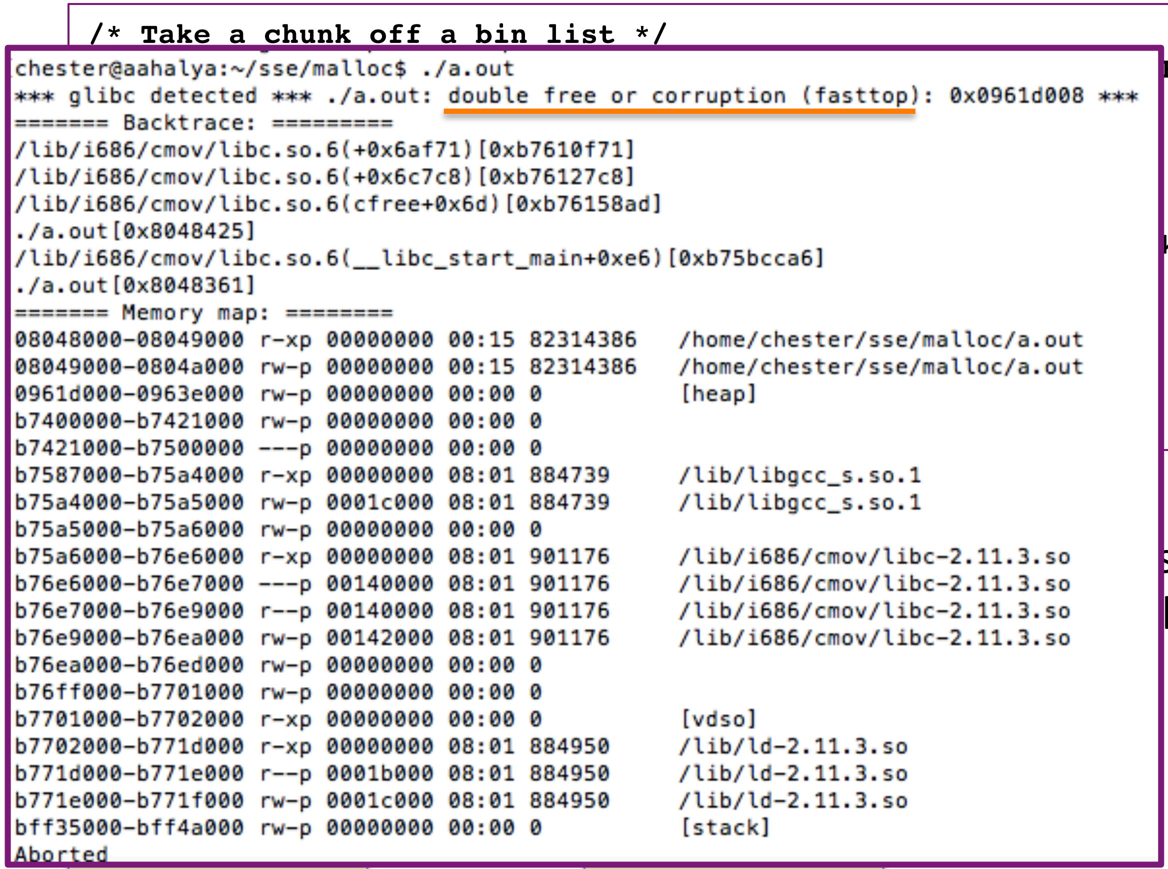


Causing programs like this to crash

```
void main()
{
      char *a = malloc(10);
      free(a);
      free(a);
}
```

Some double frees are detected

```
/* Take a chunk off a bin list */
FD)
ked list",P);
```



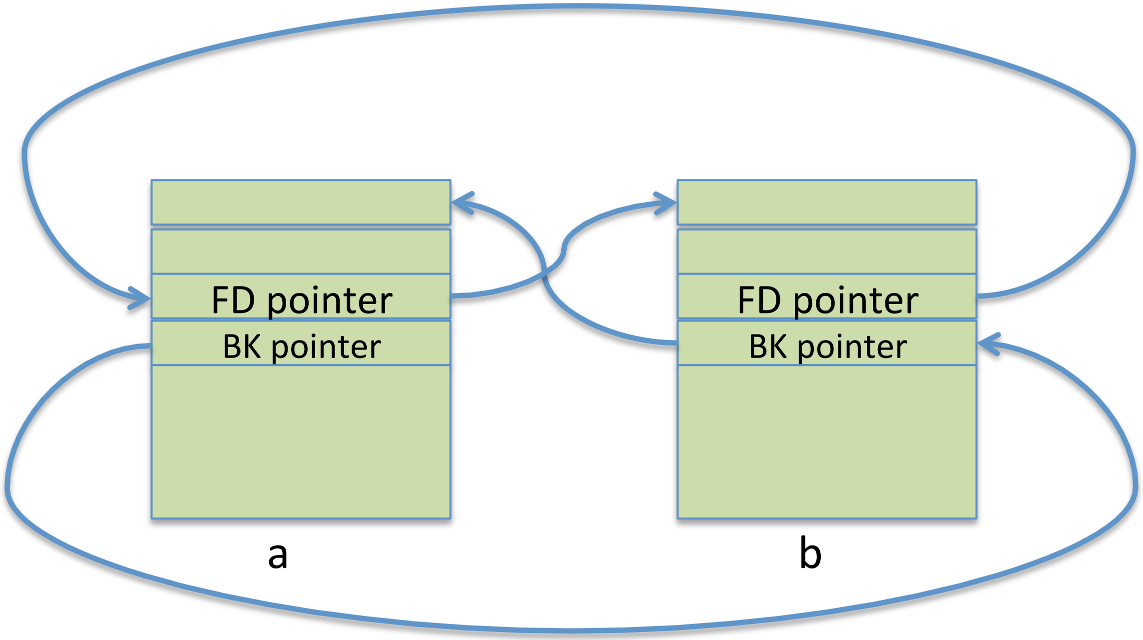
```
chester@aahalya:~/sse/malloc$ ./a.out
*** glibc detected *** ./a.out: double free or corruption (fasttop): 0x0961d008 ***
======= Backtrace: =========
/lib/i686/cmov/libc.so.6(+0x6af71)[0xb7610f71]
/lib/i686/cmov/libc.so.6(+0x6c7c8)[0xb76127c8]
/lib/i686/cmov/libc.so.6(cfree+0x6d)[0xb76158ad]
./a.out[0x8048425]
/lib/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0xb75bcca6]
./a.out[0x8048361]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:15 82314386   /home/chester/sse/malloc/a.out
08049000-0804a000 rw-p 00000000 00:15 82314386   /home/chester/sse/malloc/a.out
0961d000-0963e000 rw-p 00000000 00:00 0          [heap]
b7400000-b7421000 rw-p 00000000 00:00 0
b7421000-b7500000 ---p 00000000 00:00 0
b7587000-b75a4000 r-xp 00000000 08:01 884739     /lib/libgcc_s.so.1
b75a4000-b75a5000 rw-p 0001c000 08:01 884739     /lib/libgcc_s.so.1
b75a5000-b75a6000 rw-p 00000000 00:00 0
b75a6000-b76e6000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76e6000-b76e7000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76e7000-b76e9000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76e9000-b76ea000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b76ea000-b76ed000 rw-p 00000000 00:00 0
b76ff000-b7701000 rw-p 00000000 00:00 0
b7701000-b7702000 r-xp 00000000 00:00 0          [vdso]
b7702000-b771d000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b771d000-b771e000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b771e000-b771f000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bff35000-bff4a000 rw-p 00000000 00:00 0          [stack]
Aborted
```

sing programs like this to
h

```
void main()
{
        char *a = malloc(10);
        free(a);
        free(a);
}
```

Most double frees are not detected

After the second free

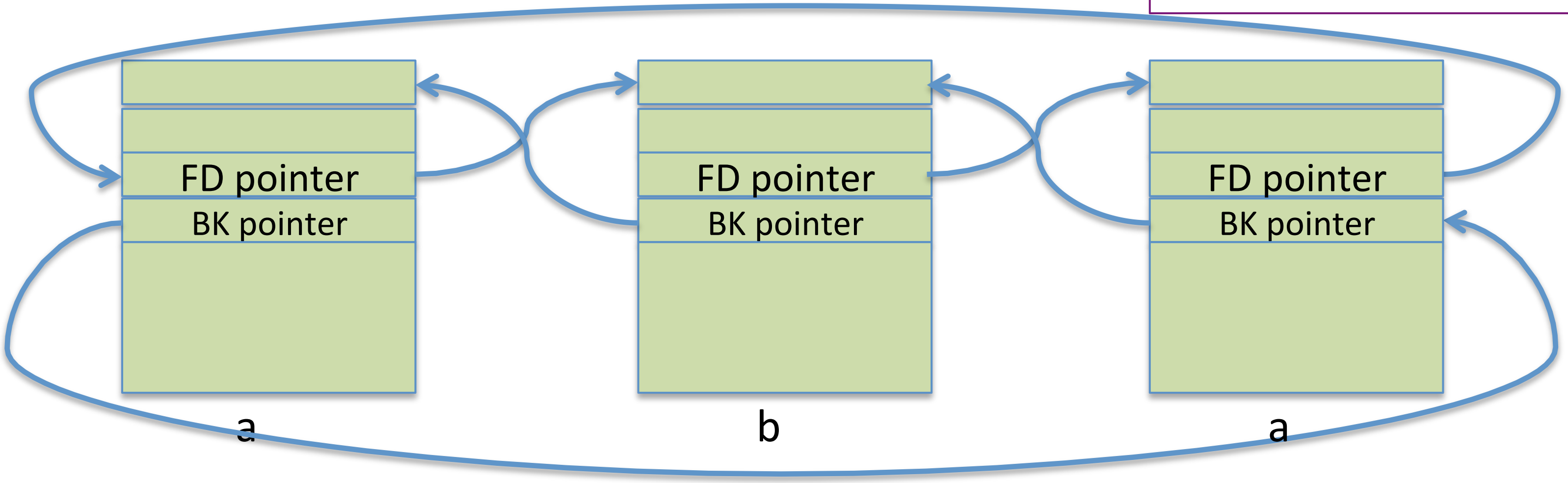


```
void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        free(a);
        free(b);
        free(a);
        printf(“The end!\n”);
}
```

Most double frees are not detected

After the third free

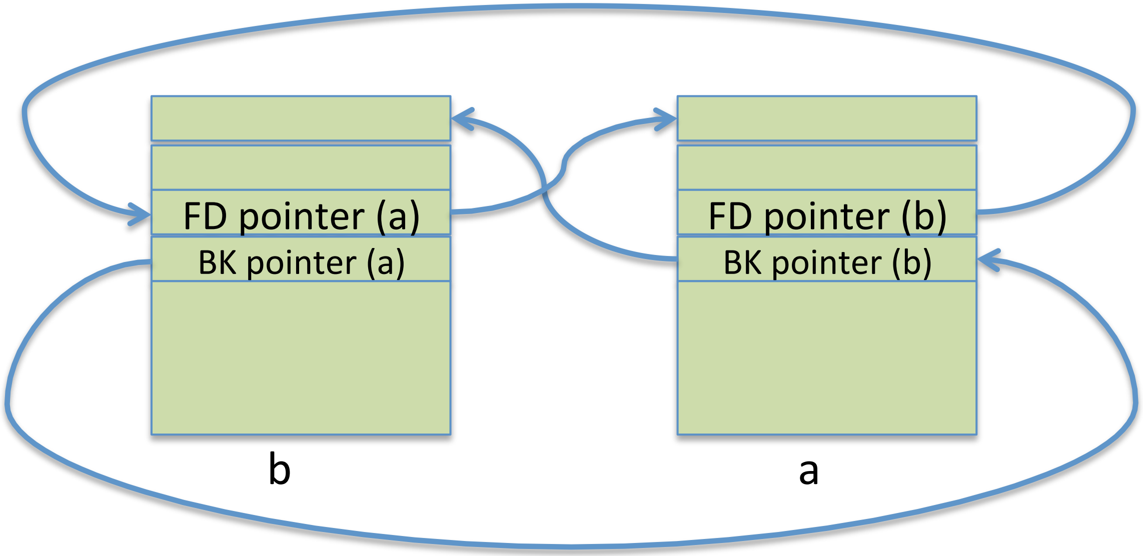
```
void main()
{
	char *a = malloc(10);
	char *b = malloc(10);
	free(a);
	free(b);
	free(a);
	printf("The end!\n");
}
```

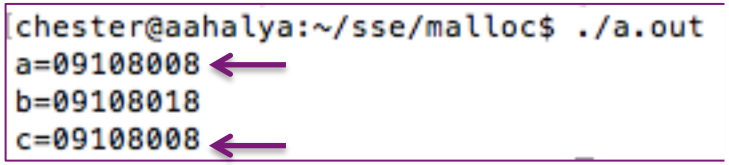


Another malloc

Another malloc
c gets allocated the same address as a

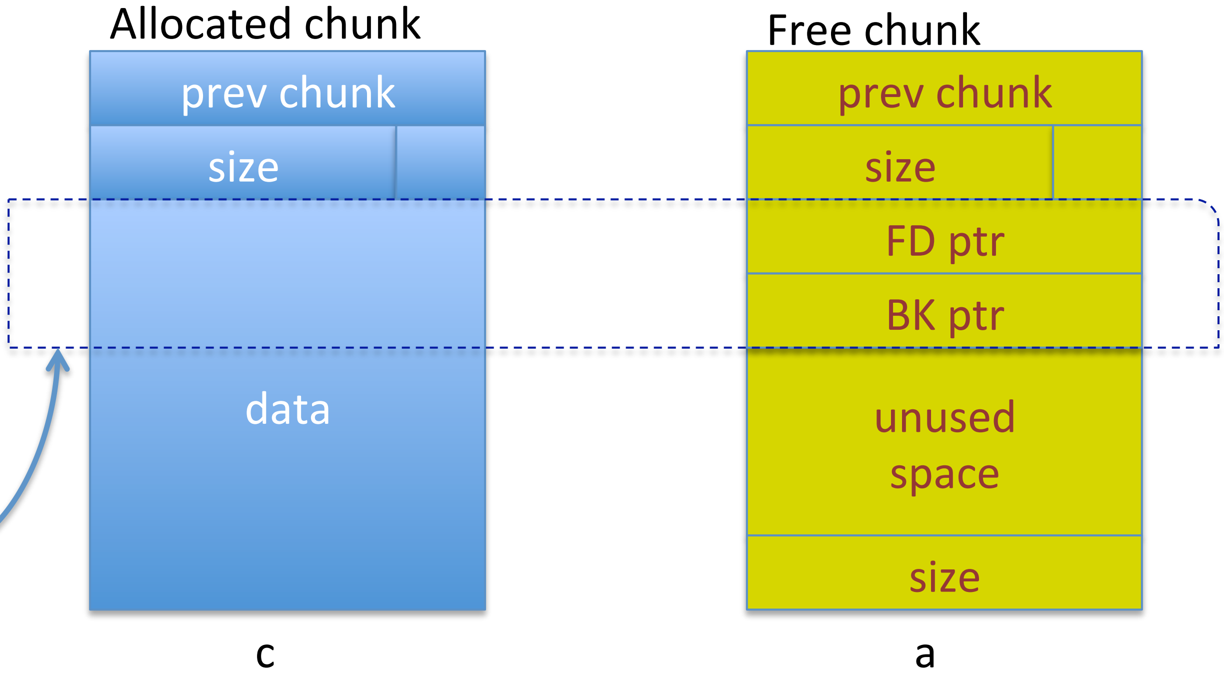
```
void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;
        free(a);
        free(b);
        free(a);
        c = malloc(10);
}
```





```
chester@aahalya:~/sse/malloc$ ./a.out
a=09108008
b=09108018
c=09108008
```

Two views of the same chunk



```
*c = 0xdeadbeef;
*(c+4) = 0xdeadbeef;
```

you can control the FD ptr and BK ptr contents using c

Exploiting

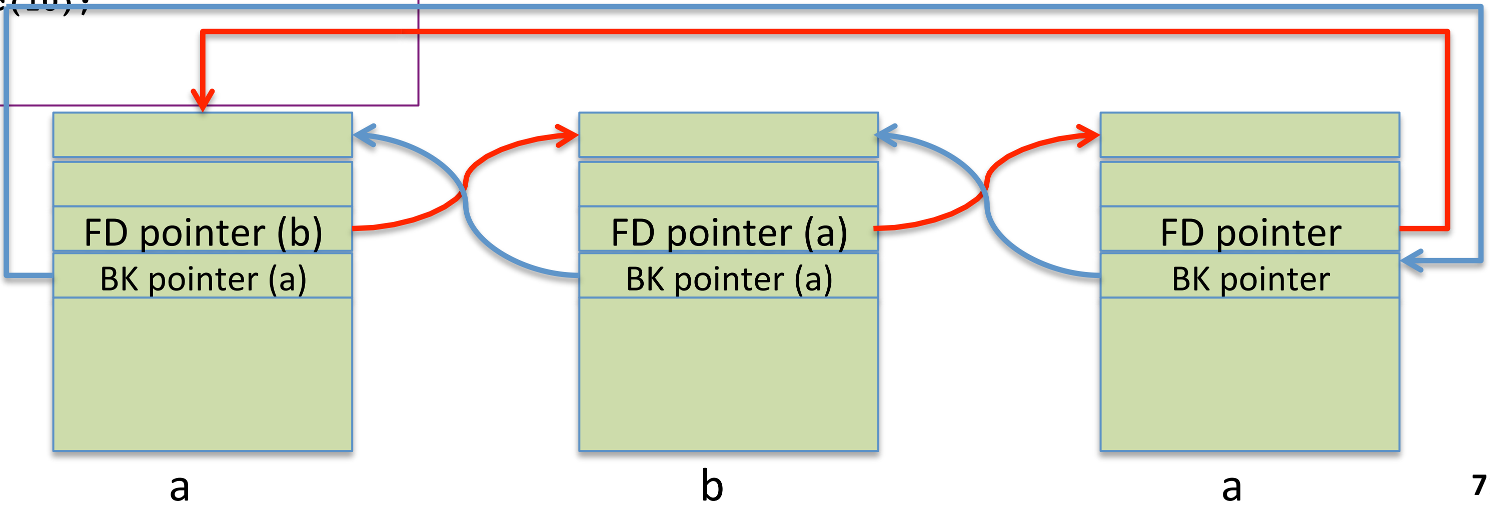
```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;

        fun1();
        free(a);
        free(b);
        free(a);
        c = malloc(10);
        *(c + 0) = GOT entry – 12 for fun1;
        *(c + 4) = payload;
        some malloc(10);
        fun1();
}
```

Need to lookout for programs that have (something) like this structure

We hope to execute payload instead of the 2nd invocation of fun1();

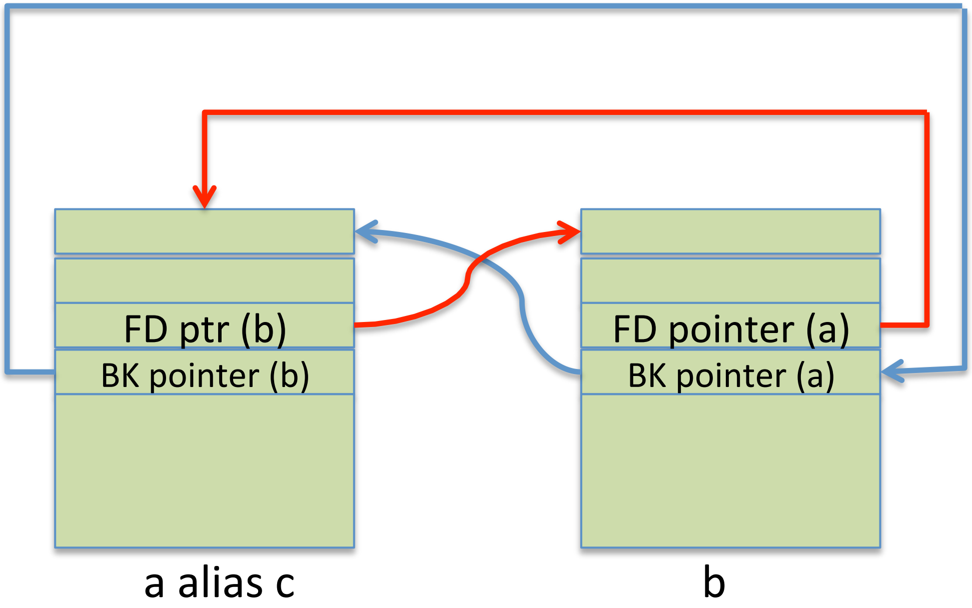
Exploiting

```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;

        fun1();
        free(a);
        free(b);
→       free(a);
        c = malloc(10);
        *(c + 0) = GOT entry for fun1;
        *(c + 4) = payload;
        some malloc(10);
        fun1();
}
```



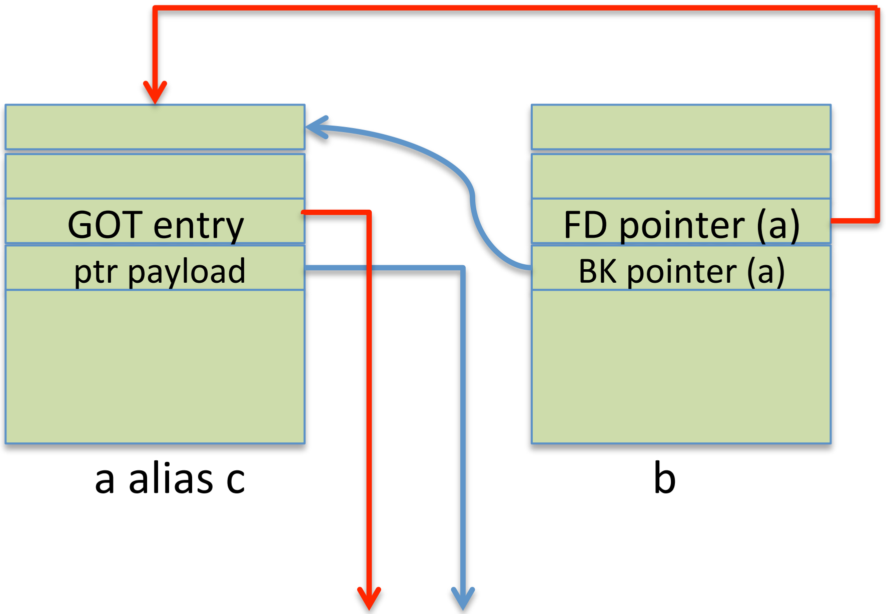
Exploiting

```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;

        fun1();
        free(a);
        free(b);
        free(a);
→       c = malloc(10);
        *(c + 0) = GOT entry for fun1;
        *(c + 4) = payload;
        some malloc(10);
        fun1();
}
```



Exploiting

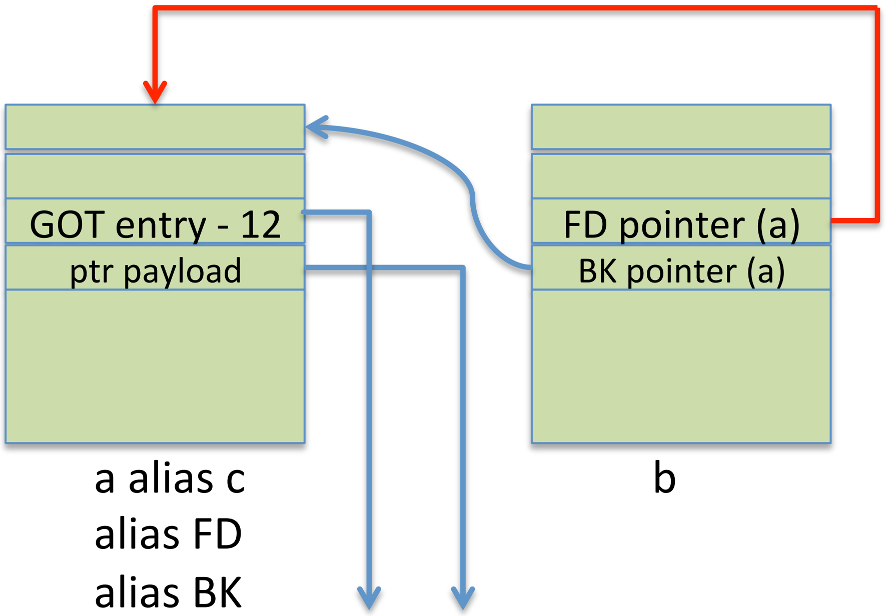
```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;

        fun1();
        free(a);
        free(b);
        free(a);
        c = malloc(10);
        *(c + 0) = GOT entry for fun1 – 12;
        *(c + 4) = payload;
        some malloc(10);
        fun1();
}
```



Exploiting

```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
        char *a = malloc(10);
        char *b = malloc(10);
        char *c;

        fun1();
        free(a);
        free(b);
        free(a);
        c = malloc(10);
        *(c + 0) = GOT entry for fun1 – 12;
        *(c + 4) = payload;
→       some malloc(10);
        fun1();
}
```

```
unlink(P){
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```



Exploiting Heap

```
char payload[] =
“\x33\x56\x78\x12\xac\xb4\x67”;

Void fun1(){}

void main()
{
      char *a = malloc(10);
      char *b = malloc(10);
      char *c;

      fun1();
      free(a);
      free(b);
      free(a);
      c = malloc(10);
      *(c + 0) = GOT entry for fun1 – 12;
      *(c + 4) = payload;
      some malloc(10);
      fun1();
}
```

Payload executes

Other heap based attacks

- Heap overflows
- Heap spray
- Use after free
- Metadeta exploits