OS Security: Other Exploits

Buffer Overread, Integer Overflow, Format String Vulnerability, Heap Overflow

Buffer Overread

Buffer Overreads

```
char pubData[] = "Public data";
char prData[] = "Some Secret Data";
int main(int argc, char **argv) {
    int i=0;
    int len=atoi(argv[i]);
    while(i < len) {
        printf("%c", pubData[i]);
        i++;
    }
    printf("\n");
}
```

What is wrong with this program?

Any vulnerability that can be exploited?

Buffer Overreads

```
char pubData[] = "Public data";
char prData[] = "Some Secret Data";
int main(int argc, char **argv) {
    int i=0;
    int len=atoi(argv[i]);
    while(i < len) {
        printf("%c", pubData[i]);
        i++;
    }
    printf("\n");
}
```

Vulnerability: program not checking any bounds on buffer read?

Parameter len is decided by user input. What if len > 12?

Can this be exploited by user?

Buffer Overreads

```
char pubData[] = "Public data";
char prData[] = "Some Secret Data";
int main(int argc, char **argv) {
    int i=0;
    int len=atoi(argv[i]);
    while(i < len) {
        printf("%c", pubData[i]);
        i++;
    }
    printf("\n");
}
```

Exploit is based on the way compiler assigns address to identifiers.

Consecutive addresses assigned to identifiers declared one after another.

prData[] address space just adjacent to that of pubData[]

Buffer Overreads

```
char pubData[] = "Public data";
char prData[] = "Some Secret Data";
int main(int argc, char **argv) {
    int i=0;
    int len=atoi(argv[i]);
    while(i < len) {
        printf("%c", pubData[i]);
        i++;
    }
    printf("\n");
}
```

If len > 12, printf in while loop shall continue to read pubData[12], pubData[13] ….

If prData[] is stored just after pubData[], this shall lead to reading data from prData[].

Data exfilteration through buffer overread.

Example: Bleeding Heart (SSL Vulnerability)

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.

Integer Overflow

Integer Overflow: Example 1

```
int main(int argc, char **argv)
{
    unsigned short int s;
    char buf[80];
    int k=atoi(argv[1]);
    s = k;
    if(s > 80) {
        printf("Attempt to buffer
        overflow\n");
        exit(0);
    }
    memcpy(buf, argv[2], k);
    buf[k] = '\0';
    return 0;
}
```

What is wrong with this program?

Any vulnerability that can be exploited?

memcpy(destination address, source address, number of bytes to be copied)

Integer Overflow: Example 1 (contd.)

```
int main(int argc, char **argv)
{
    unsigned short int s;
    char buf[80];
    int k=atoi(argv[1]);
    s = k;
    if(s > 80) {
        printf("Attempt to buffer
        overflow\n");
        exit(0);
    }
    memcpy(buf, argv[2], k);
    buf[k] = '\0';
    return 0;
}
```

Program is checking limits to avoid buffer overflow.

Is it sufficient?

memcpy(destination address, source address, number of bytes to be copied)

Integer Overflow: Example 1 (contd.)

```
int main(int argc, char **argv)
{
    unsigned short int s;
    char buf[80];
    int k=atoi(argv[1]);
    s = k;
    if(s > 80) {
        printf("Attempt to
        overread\n");
        exit(0);
    }
    memcpy(buf, argv[2], k);
    buf[k] = '\0';
    return 0;
}
```

Identifier s is unsigned short in range of [0..65535].

Identifier k is signed integer and is assigned a value supplied by user.

What if k > 65535?

Integer Overflow: Example 1 (contd.)

```
int main(int argc, char **argv)
{
    unsigned short int s;
    char buf[80];
    int k=atoi(argv[1]);
    s = k;
    if(s > 80) {
        printf("Attemt to
        overread\n");
        exit(0);
    }
    memcpy(buf, argv[2], k);
    buf[k] = '\0';
    return 0;
}
```

If k > 65535, s shall be truncated. For k = 65536, 65537, 65538 …, s = 0, 1, 2 respectively.

So buffer bound check can be successfully bypassed.

Integer Overflow: Example 1 (contd.)

```
int main(int argc, char **argv)
{
    unsigned short int s;
    char buf[80];
    int k=atoi(argv[1]);
    s = k;
    if(s > 80) {
        printf("Attempt to
        overread\n");
        exit(0);
    }
    memcpy(buf, argv[2], k);
    buf[k] = '\0';
    return 0;
}
```

Buffer overflow is possible.

Integer Overflow: Example 2

```
int main(int argc, char **argv)
{
    int  mask, k;
    mask = 0x40000000;
    k = mask + 0xC0000000;
    k = mask * 0x4;
    k = 1 - 0xFFFFFFFF;
}
```

In all these cases, k becomes zero.

May lead to unexpected results if the programmer does not take arithmetic overflow into consideration.

Integer Overflow: Example 2 (contd.)

```
int main(int argc, char **argv)
{
    int  *buf, k;
    k = atoi(argv[1]);
    buf = malloc(k* sizeof(int));
    ….
}
```

Ideally, buffer should be able to accommodate k integers.

But k*sizeof(int) may result in integer overflow resulting in a value less than expected.

Buffer can accommodate less than k integers. May result in heap/buffer overflow.

Integer Overflow: Example 3

```
int main(int argc, char **argv)
{
    int  buf[100], k;
    k = atoi(argv[1]);
    if(k > 100) {
       printf("Error!!!\n");
       exit(0);
    }
    memcpy(buf, argv[2], k);
    ….
}
```

Here, k is signed int. If too large value specified, its sign bit is set to 1 and it shall be interpreted as a negative number bypassing bounds check.

In memcpy, it is treated as unsigned and buf may overflow.

Integer Overflow: Example 4

```
int table[100];

int SetValue(int index, int
value) {
    if(index > 100) {
       return -1;
    }
    table[index] = value;
    return 0;
}
```

If index is negative, bounds check is bypassed.
Some (unintended) memory address gets overwritten.

Memory corruption..

What is some pointer address is overwritten?

Integer Overflow: Example 5

```
int SetBuffer(int sock, int *out,
int len) {
char buf1[100], buf2[100];
unsigned int size1, size2;
if(recv(sock, buf1, 100) < 0) {
exit(0);
}
if(recv(sock, buf2, 100) < 0) {
exit(0);
}
memcpy(&size1, buf1, sizeof(int));
memcpy(&size2, buf2, sizeof(int));
size = size1+size2;
if(size > len) {
    exit(0);
}
memcpy(out, buf1, size1);
memcpy(out+size1, buf2, size2);
return size;
}
```

What does this program do?

Copies contents of buf1 and buf2 into out and returns the size of out. This size is sum of sizes of buf1 and buf2 respectively.

Contents of buf1 and buf2 are set by packets received through recv.

First byte of packet is length of packet.

Integer Overflow: Example 5 (contd).

```
int SetBuffer(int sock, int *out,
int len) {
char buf1[100], buf2[100];
unsigned int size1, size2;
if(recv(sock, buf1, 100) < 0) {
exit(0);
}
if(recv(sock, buf2, 100) < 0) {
exit(0);
}
memcpy(&size1, buf1, sizeof(int));
memcpy(&size2, buf2, sizeof(int));
size = size1+size2;
if(size > len) {
    exit(0);
}
memcpy(out, buf1, size1);
memcpy(out+size1, buf2, size2);
return size;
}
```

Contents of packet are decided by user.

Both size1 and size may be in requisite limits. But sum of these two may be negative owing to integer overflow. Bounds check is a success.

Contents copied into out but size of out is less than what is being written.

May result in overflow in callee function.

Home Assignment

- Bleeding Hearts vulnerability; provide details.
- Stagefright vulnerability in Android is an integer overflow vulnerability. Provide details on how is this exploited and to what end?

String Format Vulnerability

String Format Vulnerability

```
int main(int argc, char **argv)
{
    int  a, b, c;
    // Read input
    ….
    printf("%d %d %d\n", a,b,c);
    ….
}
```

Stack contents

c
b
a
Format string "%d %d %d"
Return address
Previous frame pointer
Local variables

For every format specifier, one value is read off the stack

String Format Vulnerability (contd.)

```
int main(int argc, char **argv)
{
    int  a, b, c;
    // Read input
    ….
    printf("%d %d %d\n", a,b);
    ….
}
```

Stack contents

b
a
Format string "%d %d %d"
Return address
Previous frame pointer
Local variables

What happens when this format string is processed?

String Format Vulnerability (contd.)

```
int main(int argc, char **argv)
{
    int  a, b, c;
    // Read input
    ….
    printf("%d %d %d\n", a,b);
    ….
}
```

Compiler can not check for its consistency as
Format string be supplied at run time
Compiler needs to be aware of library function internals. Not desired as different versions of library may exist.
Checking within printf: possible.

Stack contents

b
a
Format string “%d %d %d”
Return address
Previous frame pointer
Local variables

- printf(“%s %s %s”)
- printf(“%x %x%x”)

Home Assignment

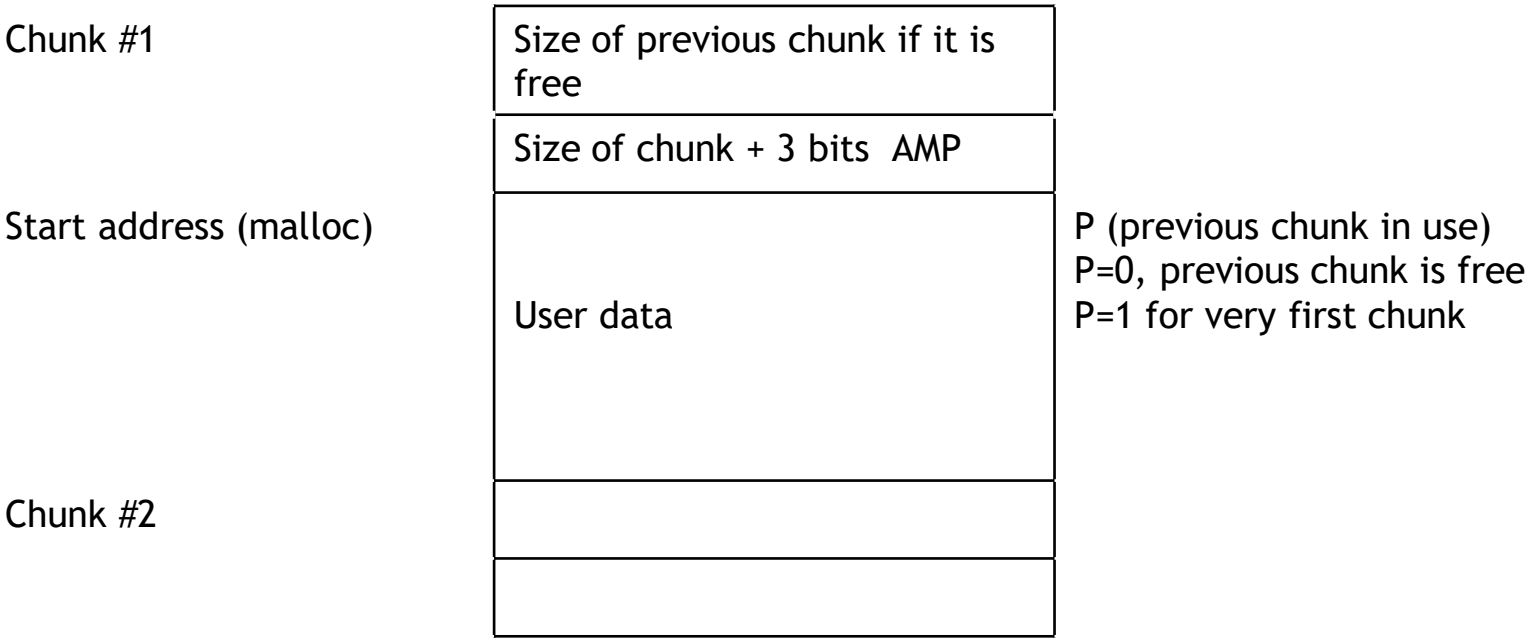
- What are format specifiers in printf/scanf … funcions?
- How can format string specifiers be used to write to an arbitrary location? [Hint: %n format specifiers] How can this be exploited to write any arbitrary value at a given location?

Heap Overflow

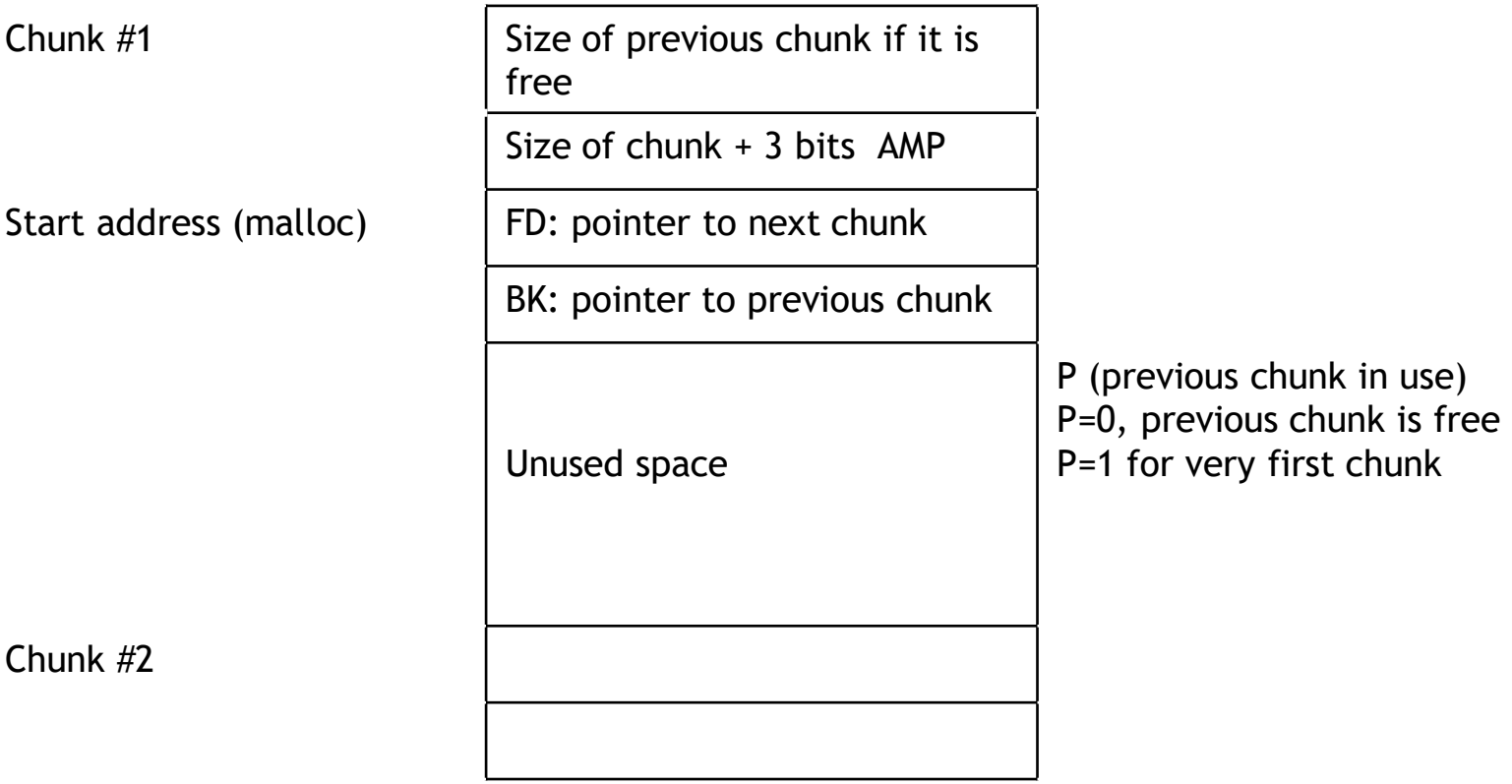
Heap Overflow Vulnerability

- Heap is organized linked list(s) of free blocks. This link information as well as size is part of heap chunk.
- OS keeps information on if a heap chunk is occupied or not?
- Heap overflow vulnerability targets
  - overwriting these pointer information: corruption of heap data
  - Gaining access to libc
  - Overwriting function pointers in heap data for arbitrary code execution
- Needs extensive knowledge of how heap is organized by OS

Allocated (in use) Heap Chunk



Free Heap Chunk



Unlinking

- A heap chunk node is removed from free list
- Here node P is being unlinked; prev and next are previous and next heap chunks (free) for p

```
void unlink(chunk *P, chunk *prev, chunk *next) {
    next = P->FD;
    prev = P->BK;
    if(next)
        next->BK = prev;
    if(prev)
        prev->FD = next;
}
```

Double Free

- What happens with this code?

```
void main() {
     char *a = malloc(10);
     free(a);
     free(a);
}
void unlink(chunk *P, chunk *prev, chunk *next) {
    next = P->FD;
    prev = P->BK;
    if(next  && (next->BK != P))
          error
    If(prev && (prev->FD != P)
          error
    next->BK = prev;
    prev->FD = next;
}
```

Double Free

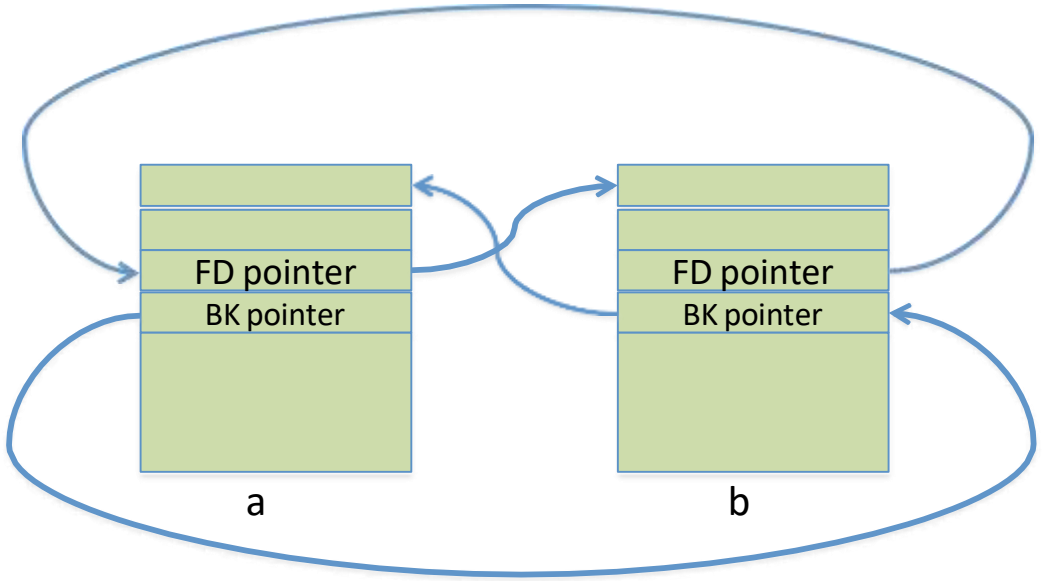
- What happens with this code?

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

Use After Free

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

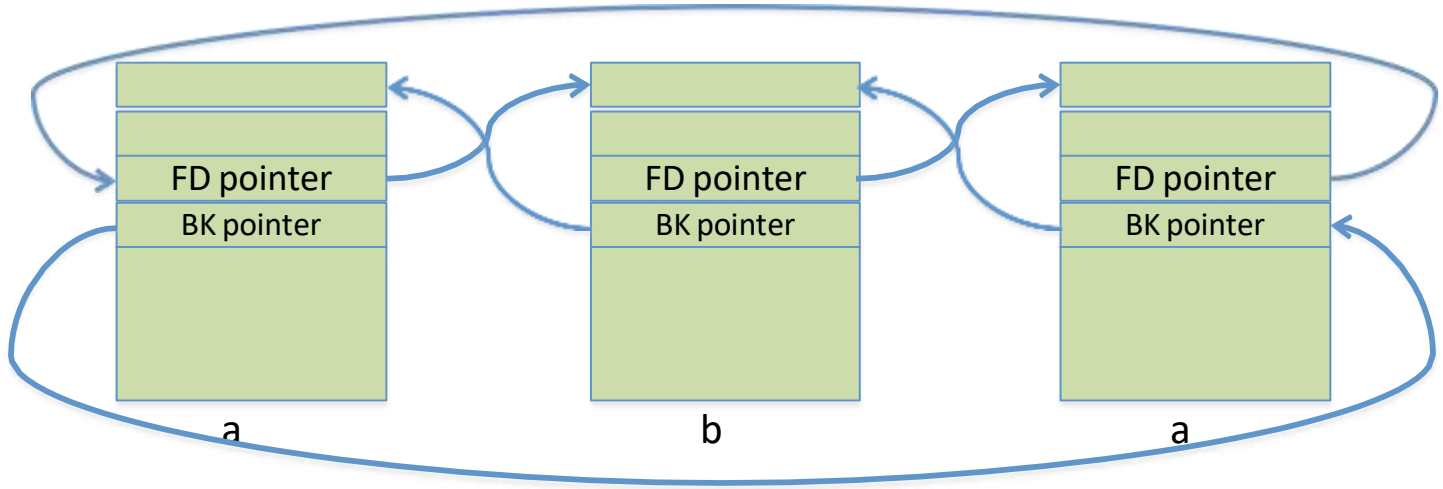
After second free



Use After Free

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

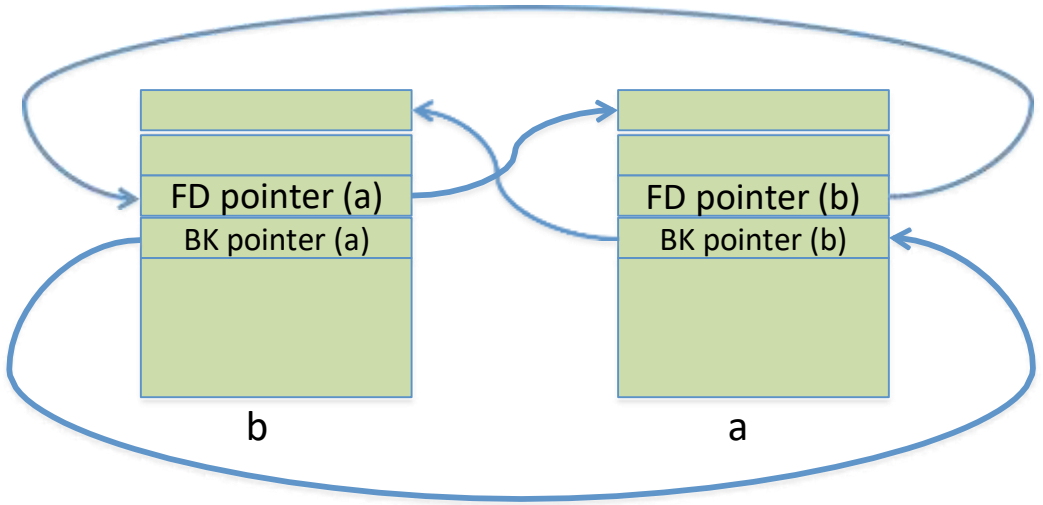
After third free



Use After Free

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

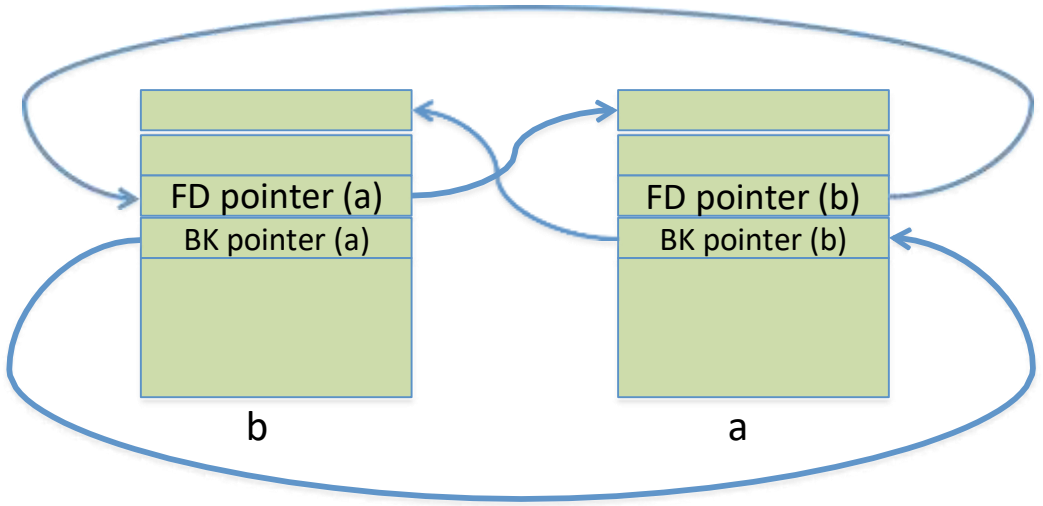
Identifier c gets allocated the same address as a



Use After Free

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

Identifier c gets allocated the same address as a
Can be used to print what a contains



OS perspective

Allocated chunk
Prev chunk
Size
*(c+0) *(c+4) data
c

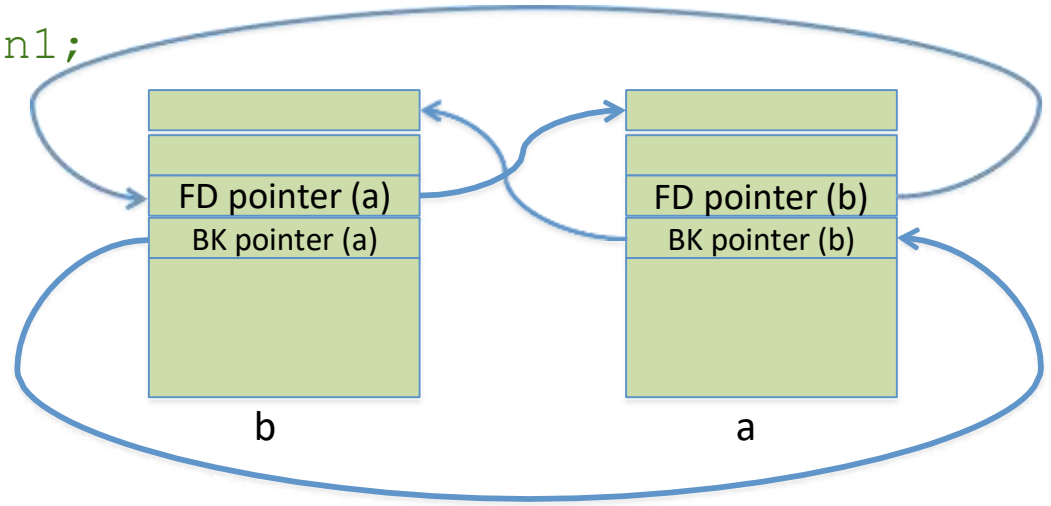
Free chunk
Prev chunk
Size
FD ptr
BK ptr
Unused Space
size
a

Same chunk is allocated as well as free from OS perspective..
Writing to heap through c is modifying FD and BK pointer of a

Multiple Free

```
void main() {
    char *a = malloc(10);
    char *b = malloc(10);
    fun1();
    free(a);
    free(b);
    free(a);
    char *c = malloc(10);
    *(c+0) = GOT entry for fun1;
    *(c+4) = payload;
    malloc(); // allocates b
    fun1();
}
```

Corrupts pointers of a.



Data Leakage

- Multiple free lists exist (decided by size of chunk); each free list called a bin
  Unlinking policies are different for different bins; some return free node to head/tail; some combine free chunks to create a larger chunk
- FD and BK of a free chunk at the head or tail of a bin contains libc-addresses.
- Can be used to leak libc-address through a heap-based vulnerability.
  - Use After Free (UAF) vulnerability: we could just free a chunk and then print its data. First 8 bytes of data are the FD, we would get a libc address right away.
  - If the heap metadata gets corrupted in such a way that two chunks are overlapping in memory, we can free one chunk (which now contains libc-addresses) and then use the other chunk to print those addresses.

More on Heap Vulnerability

- https://github.com/shellphish/how2heap

Thank you.