

## Static vs. Dynamic Variables

- This C function allocates 256 bytes of memory in the stack for a variable called 'buffer':
  - `char buffer[256];`
  - buffer's size is static and cannot change
- Memory may also be allocated dynamically on the heap during runtime via the C `new( )` and `malloc( )` functions:
  - `char *buffer = malloc(256);`
  - \*buffer's size may change during run time

3

Buffers may be static (set at compile time), or dynamic (set during runtime).

Both static and dynamic buffers may be 'smashed,' or written beyond the expected boundary. This happens when proper bounds checking is not performed by the programmer. Either form may result in an attacker seizing control of program execution

# Linux Heap Layout

- Linux heap chunk management information is stored 'in band' with user data in memory
- Writing data past the end of a chunk boundary may overwrite the next chunk's management fields
- Fields include
  - PREV\_SIZE (size of the previous chunk)
  - SIZE (size of the current chunk)
  - 'bd' and 'fd' pointers are added when the chunk is marked unallocated

5

Chunks are areas of memory in the heap that are dynamically allocated via commands such as malloc (memory allocation), and are later returned to the available memory pool via free():

Chunk format is: [ PREV\_SIZE ] [ SIZE ] [ data..... ]

The first field is PREV\_SIZE, or the size of the previous chunk. It is 4 bytes long.

The next field is the SIZE of the current chunk. It is also 4 bytes long, except the least significant bit.

The least significant bit of SIZE is used as a flag called PREV\_INUSE, which determines whether the previous chunk is unallocated.

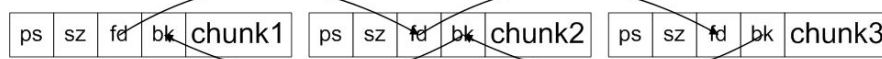
An unallocated chunk adds 2 fields: the Forward pointer (called fd) and Back pointer (called bk). These pointers are part of a doubly-linked list (forwards and backwards), which are used to consolidate unallocated heap chunks when they are free()ed. free() will remove the chunk from the linked list via the unlink() function.

Format is: [ PREV\_SIZE ] [ SIZE ] [ fd ] [ bk ] [remaining data...]

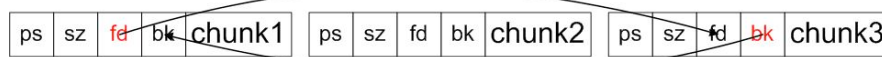
It's important to note that data in an allocated chunk begins where the fd and bk pointers are located in an unallocated chunk.

# Unlinking a Chunk

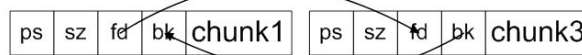
Chunks1, 2, and 3 are joined by a doubly-linked list



Chunk2 may be unlinked by rewriting 2 pointers



Chunk2 is now unlinked



6

When the chunks are no longer needed they will be marked unallocated. The `free()` function frees unallocated chunks, and calls `unlink()` to remove them from allocated memory.

Unallocated chunks are joined in a doubly-linked list. A chunk is then removed from the doubly-linked list via two writes to memory. The contents of chunk2's `fd` field will be copied to the location referenced by chunk2's `bk` field. Then the contents of chunk2's `bk` field will be copied to the location referenced by chunk2's `fd` field.

## Unlink in more detail

- The contents of chunk2's bk are copied to memory location listed in chunk2's fd
  - Contents of fd are also copied to location bk
- In other words, 'what' is copied to 'where'
- Hijacking the unlink process with fake chunk fields allows control of the 'what' and 'where'
- The attacker can write 4-bytes to virtually any location in memory!

7

For an attacker, the key feature of unlink() is the ability to write two 4-byte words to memory (the new fd and bk pointers). A fake heap chunk header which is shifted into position via a heap overflow may be used to overwrite virtually any 4-byte word in memory.

This attack uses hundreds of fake heap structures to force unlink to copy the contents of bk to fd hundreds of times. This technique is used to copy the shellcode to memory, and then overwrite a return pointer (pointing to the location of the shellcode).

Unlink also copies the contents of fd to memory location bk. This damages a portion of memory, but has no effect on the attack itself.

# Overwriting Memory

- By 'moving the goalposts', the attacker can jump to a fake chunk header in the middle of user-controlled data
- The attacker can:
  - Control the fake fd and bk pointers
  - Control the 'what' and 'where' to write
  - Write 4-bytes of data to virtually any memory location
  - Write as many times as required by using multiple unlinks

14

Leveraging the 'what' to 'where' techniques via fake chunk headers creates a 'write 4 bytes to virtually any memory location' primitive.

Unlink is called hundreds of times during the attack, overwriting large portions of memory.

# Writing 'what' to 'where'

- When the fake chunks are freed, unlink copies bk to location fd
  - bffffeeb is copied to location bfffe0be
  - bffffe15 is copied to location bfffe0bf
  - bffffe42 is copied to location bfffe0c0
  - bffffe4c is copied to location bfffe0c1
  - Etc...

						fake prev_size				fake size				fake fd				fake bk						
.	.	B	B	B	B	B	f8	ff	ff	ff	be	e0	ff	bf	eb	fe	ff	bf	B	.	.			
.	.	B	B	B	B	B	f8	ff	ff	ff	bf	e0	ff	bf	15	fe	ff	bf	B	.	.			
.	.	B	B	B	B	B	f8	ff	ff	ff	c0	e0	ff	bf	42	fe	ff	bf	B	.	.			
.	.	B	B	B	B	B	f8	ff	ff	ff	c1	e0	ff	bf	4c	fe	ff	bf	B	.	.			

15

This image zooms in on the fake heap structures embedded in the CVS content entries.

Unlink will copy each fake bk to the memory referenced in the fake fd field, in the order the chunks are unlinked.

By referencing increased memory locations in 1-byte steps, this allows an attacker to copy shellcode to memory one byte at a time.

Note that fd is also copied to location bk:

- bffffebe is copied to location bfffe0eb
- bfffe0bf is copied to location bffffe15
- bffffec0 is copied to location bfffe042
- bffffec1 is copied to location bfffe04c

This damages a portion of memory, but has no effect on the attack.

# Copy Shellcode, Byte-by-Byte

	Memory address									
	0xbfffe0be	0xbfffe0bf	0xbfffe0c0	0xbfffe0c1	0xbfffe0c2	0xbfffe0c3	0xbfffe0c4	0xbfffe0c5	0xbfffe0c6	0xbfffe0c7
Unlink #1	eb	fe	ff	bf						
Unlink #2	eb	15	fe	ff	bf					
Unlink #3	eb	15	42	fe	ff	bf				
Unlink #4	eb	15	42	4c	fe	ff	bf			
Unlink #5	eb	15	42	4c	34	fe	ff	bf		
Unlink #6	eb	15	42	4c	34	43	fe	ff	bf	
Unlink #7	eb	15	42	4c	34	43	4b	fe	ff	bf
Etc...	eb	15	42	4c	34	43	4b	48	fe	ff

16

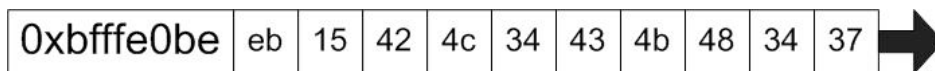
Using the 'byte-by-byte' method, an arbitrary amount of shellcode may be copied to memory.

Here is the actual shellcode used in the attack:

```
ab_shellcode[] =
"\xeb\x15\x42\x4c\x34\x43\x4b\x48\x34\x37\x20\x34\x20\x4c\x31\x46\x33"
"\x20\x42\x52\x4f\x21\x0a\x31\xc0\x50\x68\x78\x79\x6f\x75\x68\x61\x62"
"\x72\x6f\x89\xe1\x6a\x08\x5a\x31\xdb\x43\x6a\x04\x58\xcd\x80\x6a\x17"
"\x58\x31\xdb\xcd\x80\x31\xd2\x52\x68\x2e\x2e\x72\x67\x58\x05\x01\x01"
"\x01\x01\x50\xeb\x12\x4c\x45\x20\x54\x52\x55\x43\x20\x43\x48\x45\x4c"
"\x4f\x55\x20\x49\x43\x49\x68\x2e\x62\x69\x6e\x58\x40\x50\x89\xe3\x52"
"\x54\x54\x59\x6a\x0b\x58\xcd\x80\x31\xc0\x40\xcd\x80"
```

## The Result After Unlinking

- After all `unlink()`s have completed, the shellcode is copied into contiguous memory:



17

After unlink is complete, the shellcode is copied to memory.

Some memory locations cannot be referenced by the attack, such as any ending with 0x00 (null), 0x0a (newline), 0x0d (carriage return), and 0x2f (slash). These will break the CVS entry.

This creates holes in the shellcode, where a byte must be skipped to avoid referencing a 'bad byte', such as a null.

The attacker works around this limitation by adding 'jumps' to the shellcode (hex 0xeb, equivalent to assembly 'JMP'). The attack begins 0xeb 0x15, or 'jump 21 bytes (hex 15)'. Characters in between the jumps are ignored, so the attacker treats them as comments, the first is 'BL4CKH47 4 L1F3 BRO!' (0x42 0x4c 0x34 0x43 0x48 0x34 0x37...).



## Jump to the Shellcode

- After the shellcode is written to memory, use our 'what' to 'where' method to overwrite a return pointer
- Write: <location of the shellcode> to: <location of a return pointer>
- When the function exits, the program will jump to the shellcode and execute
- Game over!

18

Finally, the attacker overwrites a return pointer with the address of the shellcode.

The attacker then sends 'noop's to the CVS server, flushing out pending responses, and triggering unlinks.

Here is a summary of what will happen next:

- The fake heap chunk headers will be unlink()ed
- 'what' will be copied to 'where'
- The shellcode will be copied to memory byte-by-byte
- A return pointer will be overwritten with the location of the shellcode
- Transfer of control will be passed to the shellcode

# Summary

- The 'in band' design of the heap may place chunk management fields adjacent to user-controlled data
- A single-byte error ('Off by 1') may allow an attacker to alter these fields
- `unlink()` allows a 'write 4 bytes virtually anywhere' primitive
- glibc was patched in version 2.3.5 to address this attack

19

This attack may allow an attacker to

Copy shellcode to memory

Overwrite return pointers

Alter virtually any location in memory

glibc (GNU libc) was updated in version 2.3.5 to add a number of additional checks that make the 'write any 4 bytes in memory' `unlink()` attack ineffective.

References:

Aleph One, 'Smashing the stack for Fun and Profit'. Phrack Volume Seven, Issue Forty-Nine. URL: <http://www.phrack.org/issues.html?issue=49&id=14#article>

Anonymous, 'Once upon a free()...' Phrack Volume 0x0b, Issue 0x39. URL: <http://www.phrack.org/issues.html?issue=57&id=9#article>

cvs\_linux\_freebsd\_HEAP exploit. URL: [http://www.packetstormsecurity.org/0405-exploits/cvs\\_linux\\_freebsd\\_HEAP.c](http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c)

Nipon. "Overwriting .dtors using Malloc Chunk Corruption". The Infosec Writers Text Library, 05/09/2003. URL: <http://www.infosecwriters.com/texts.php?op=display&id=19>

vl4d1m1r "The art of Exploitation: Come back on an exploit". Phrack Volume 0x0c, Issue 0x40. URL: <http://www.phrack.org/issues.html?issue=64&id=13&mode=txt>