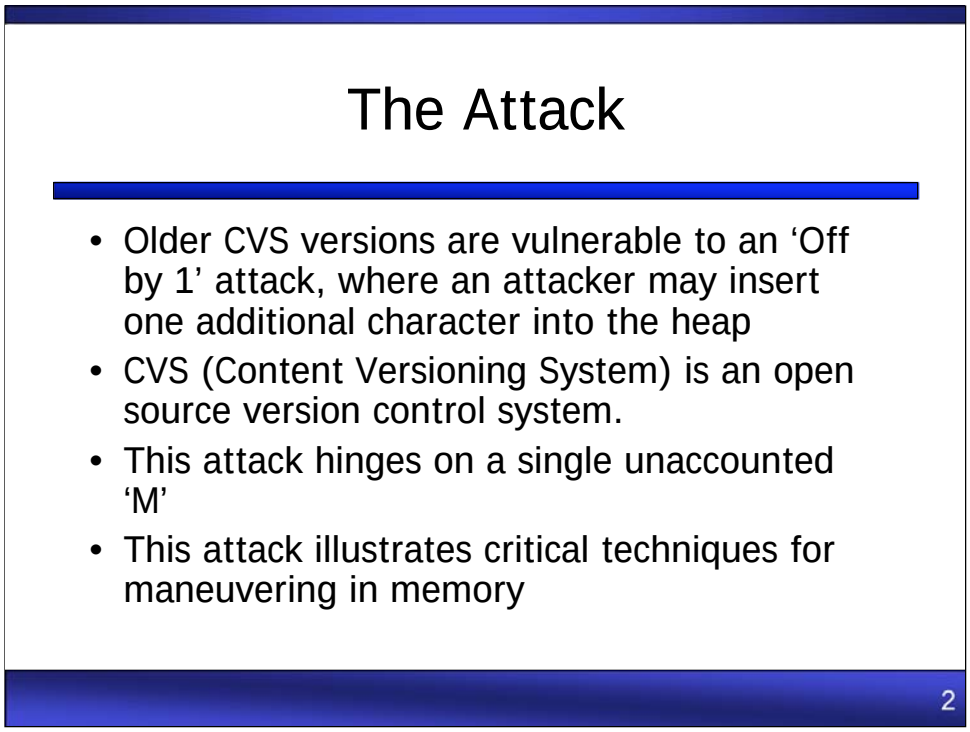
Heap ‘Off by 1’ Overflow Illustrated

Eric Conrad
October 2007

1



Vulnerable versions of CVS include stable release versions 1.11.x up to 1.11.15, and feature release versions 1.12.x up to 1.12.7.

Common Vulnerabilities and Exposures (CVE): CAN-2004-0396.

Link: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0396

Linux-based systems running glibc version lower than 2.3 may be vulnerable. Similar CVS attacks exist for FreeBSD and Solaris-based systems.

The initial research for this attack took place during the summer of 2004, and was included in my GIAC GCIH Gold paper. URL: http://www2.giac.org/certified_professionals/practicals/gcih/621.php

In May of 2007 the attack's authors wrote their own analysis, which appeared in Phrack #64. URL: http://www.phrack.org/issues.html?issue=64&id=13&mode=txt

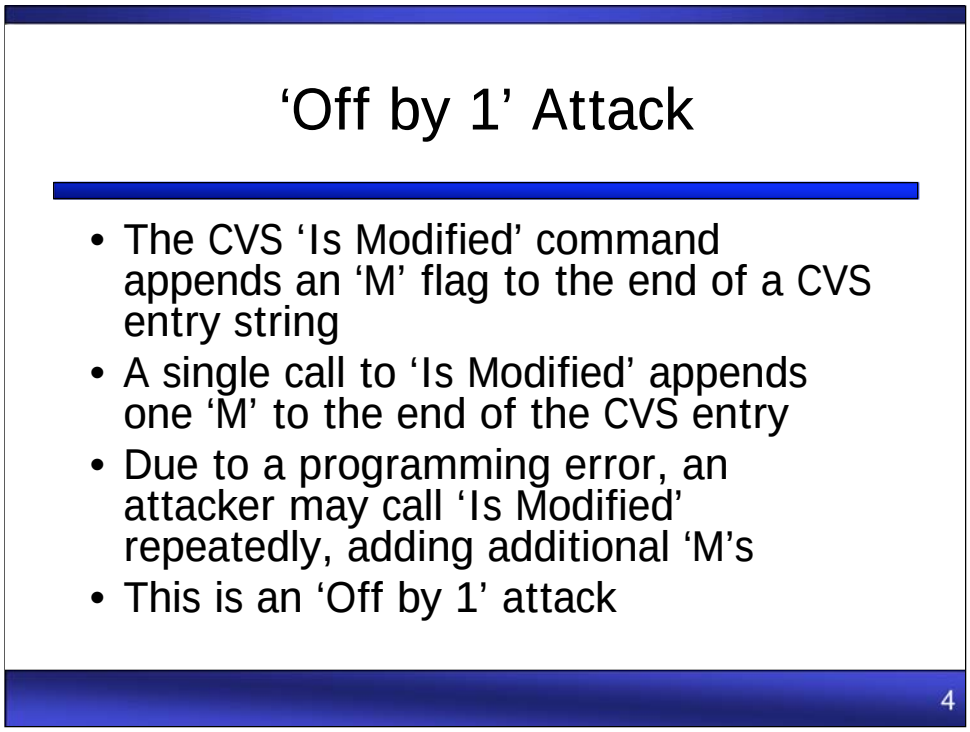
Static vs. Dynamic Variables

- 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

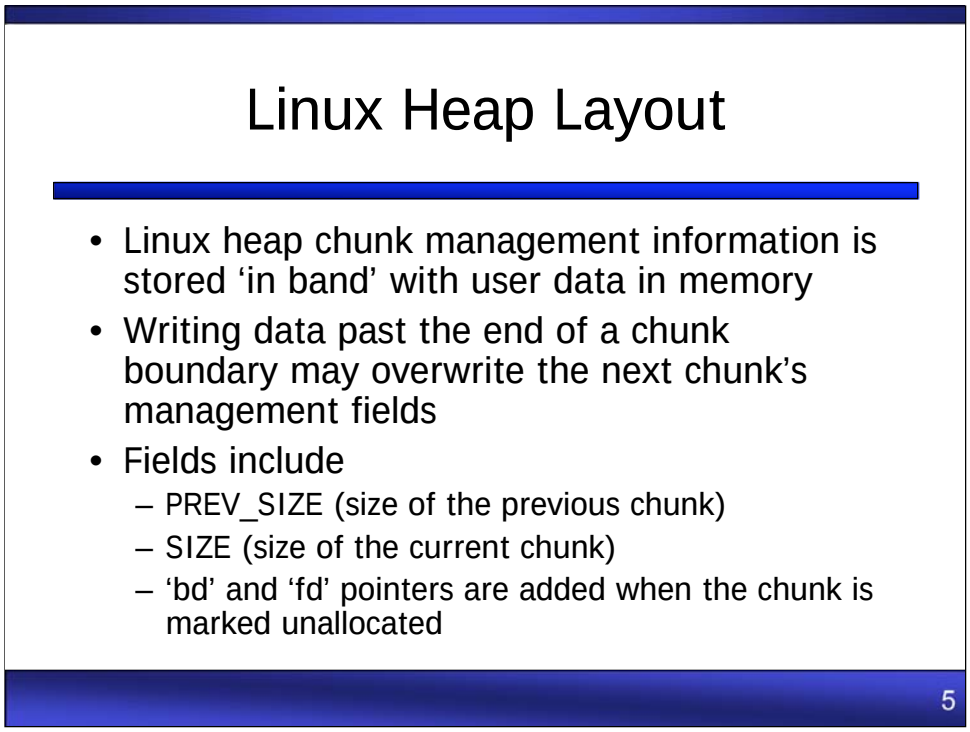
Buffers may be static (set at compile time), or dynamic (set during runtime).

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



This attack leverages an 'off-by-one' heap boundary error. The vulnerable CVS 'Is Modified' function was coded with the assumption that it would be called once for a given entry, and allocates 1 additional byte for the addition of an "M" flag (for "Modified") to the name of that entry. Repeatedly calling 'Is Modified' for the same entry results in the insertion of additional "M" flags, overflowing the heap chunk into the next allocated chunk. While an "off-by-one" technique used, an attacker may overwrite as many bytes as desired via repeated calls to 'Is Modified'.

Access to the next heap chunk allows manipulation of the next chunk's header, which may be subverted to overwrite 4-byte words in memory.



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.

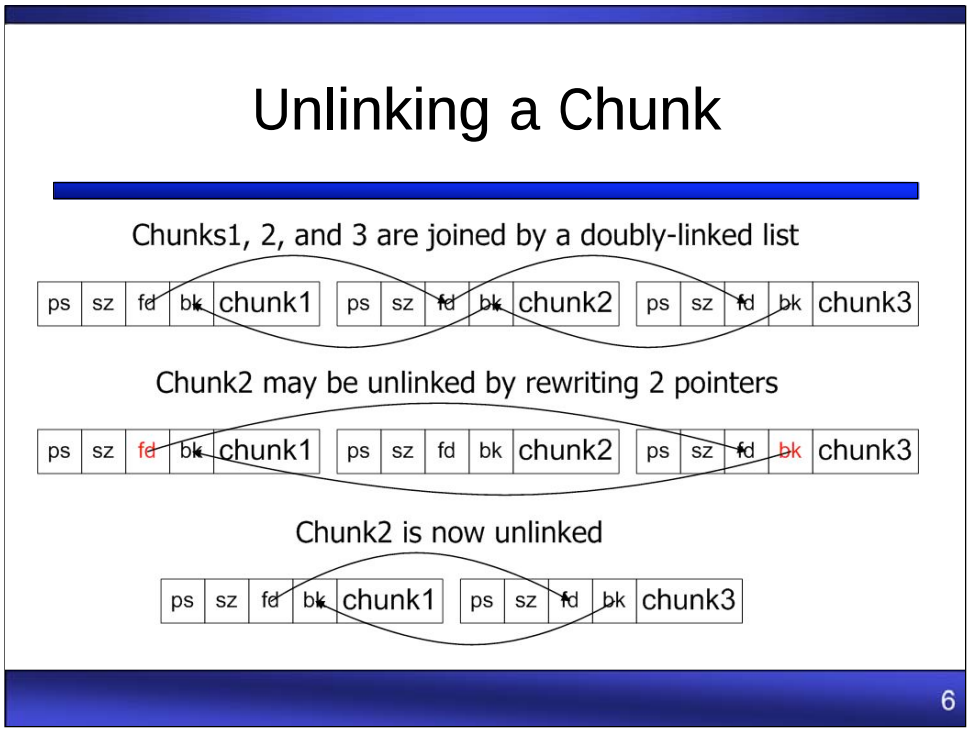
The next field is the SIZE of the current chunk. It is also 4 bytes long, except the least significant bit.

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.

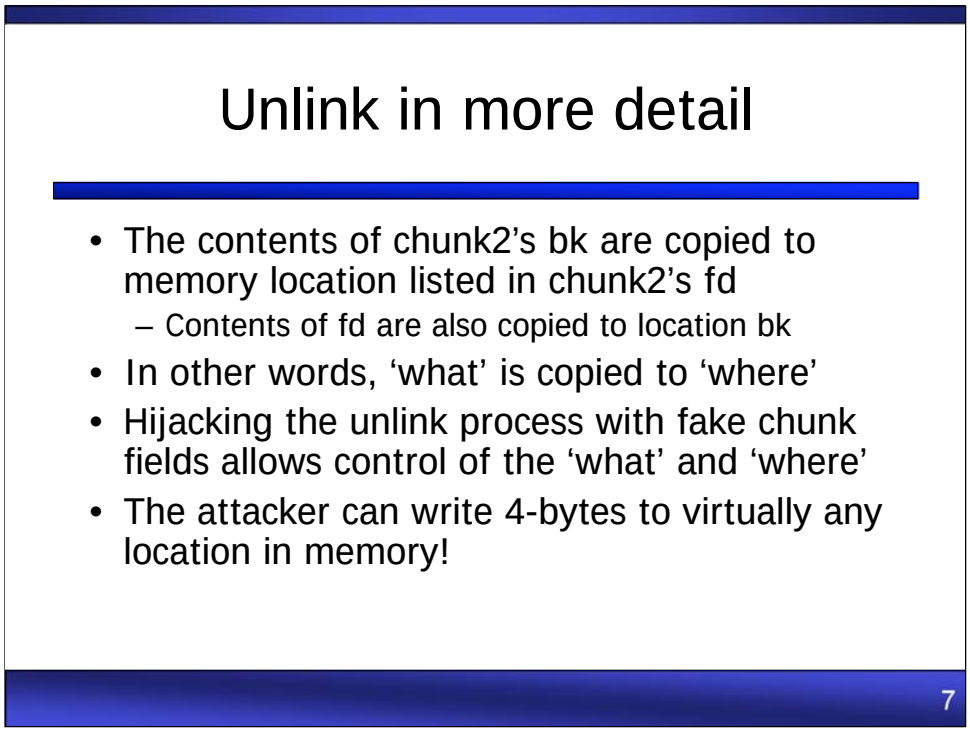
Format is: [PREV_SIZE] [SIZE] [fd] [bk] [remaining data…]

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



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.



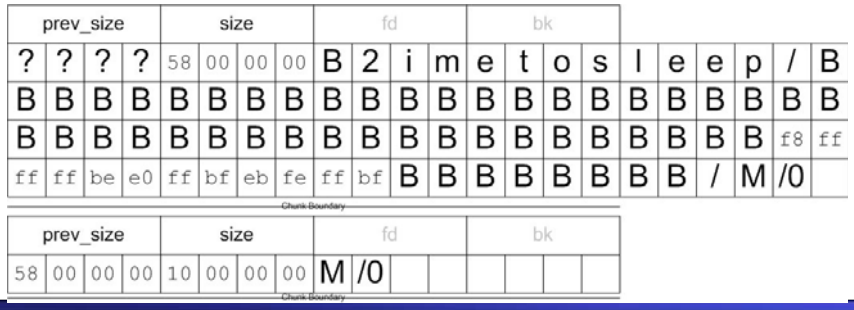
For an attacker, the key feature of unlink() is the ability to write two 4-byte words to memory (the new fd and bk pointers). A fake heap chunk header which is shifted into position via a heap overflow may be used to overwrite virtually any 4-byte word in memory.

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.

The Heap Before the Smash

- Create a CVS entry with fake chunk fields embedded in user-controlled data
- Fake fields begin 60 bytes into the data

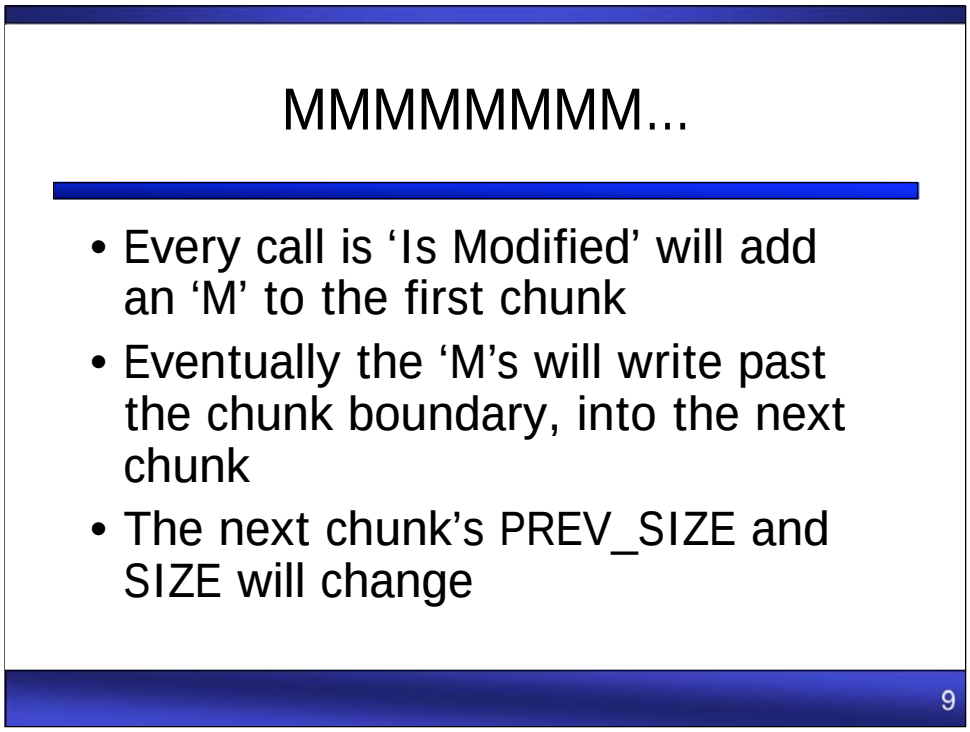


The attacker creates hundreds of CVS entries in the format “XXtimetosleep/BBBB….” (B2time… in this example).

The attacker has embedded fake chunk management fields beginning 60 bytes into the chunk. The attacker must avoid bytes that will break the string, such as null, newline, and carriage return. Otherwise, any data may be embedded.

The CVS server treats these entries as valid names of content.

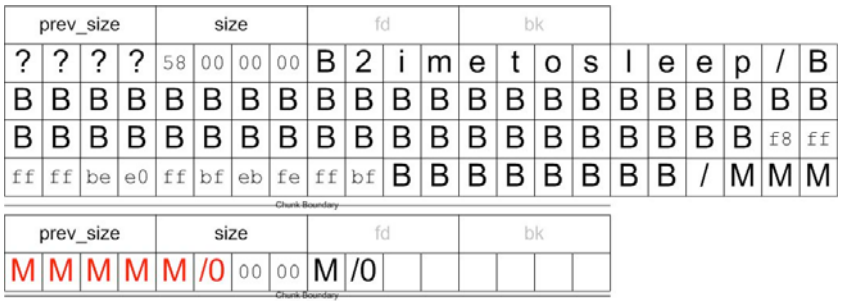
This portion of the heap alternates between 88-byte chunks (the name of the content) and 16-byte entries (allocated for a flag). 16 bytes is the minimum size of a chunk.



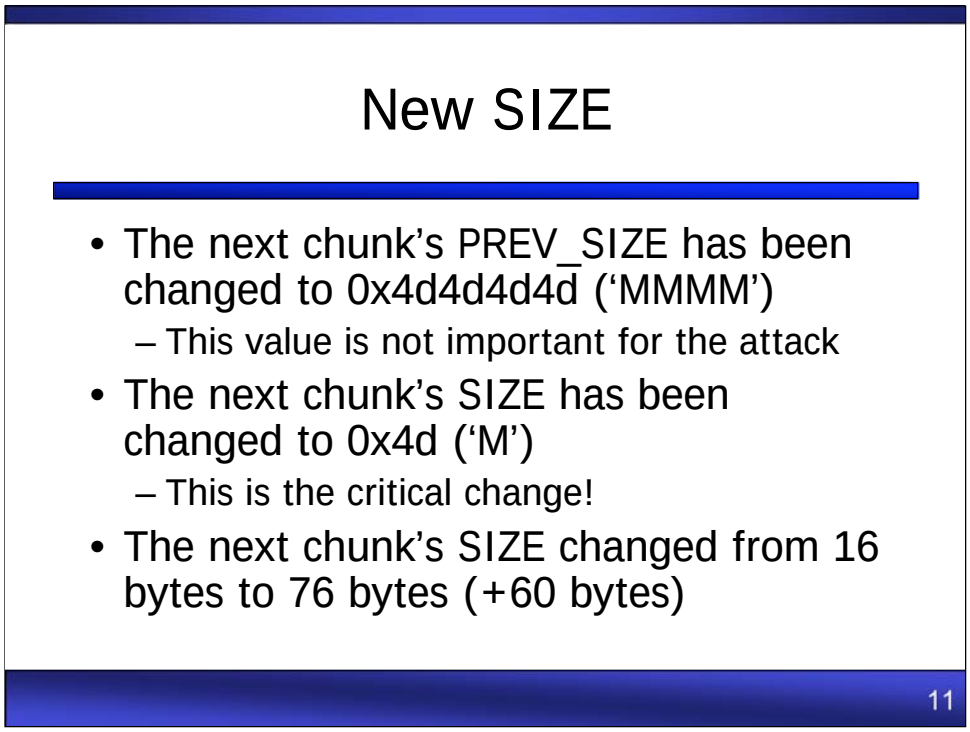
Repeatedly calling ‘Is Modified’ will append additional ‘M’ characters (with one trailing null) to the end of the CVS content name. Only the first call to ‘Is Modified’ is expected by the programmer; subsequent calls trigger the Off-by-1 vulnerability and eventually smash the chunk boundary.

The Heap After the Attack

- 'Is Modified' is called 8 times, smashing the chunk boundary
- The next chunk's PREV_SIZE and SIZE change!



The chunk boundary has been smashed, and the PREV_SIZE and SIZE fields of the next chunk have been altered.



PREV_SIZE is now 0x4d4d4d4d. The PREV_SIZE field of these chunks will not be referenced during the attack, so the value is not important.

Size has been changed from 0x10 (16 bytes) to 0x4c (76 bytes)

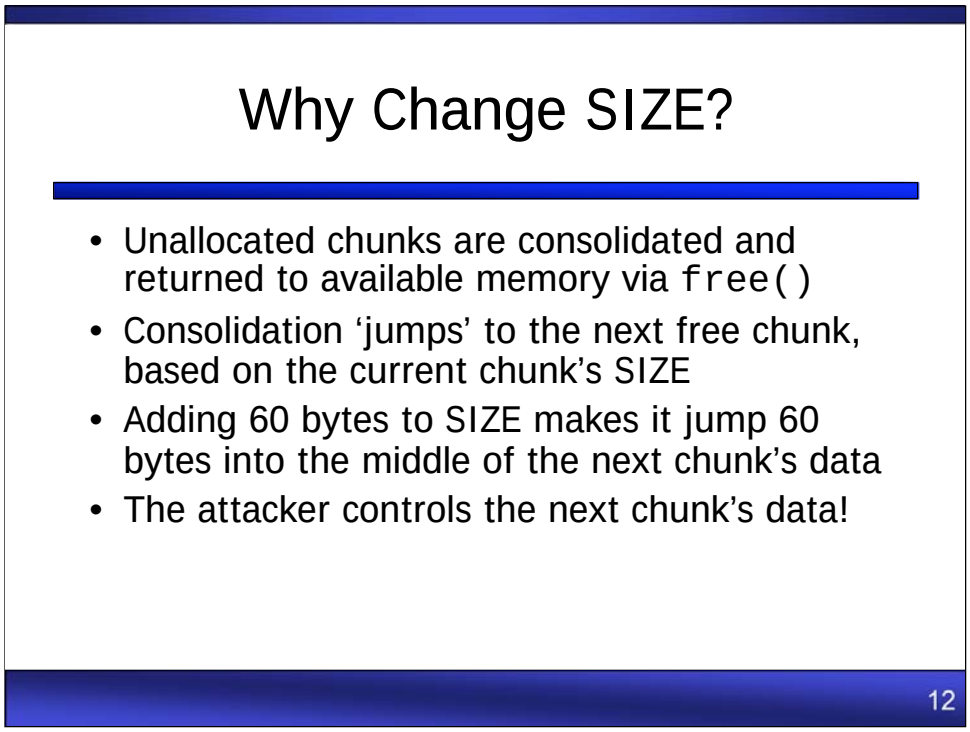
Note: the least significant bit of SIZE is the PREV_INUSE flag: that bit is not part of SIZE.

Here is a bitwise representation of an ASCII "M, the new SIZE:

01001101 == "M" == 0x4d == 77

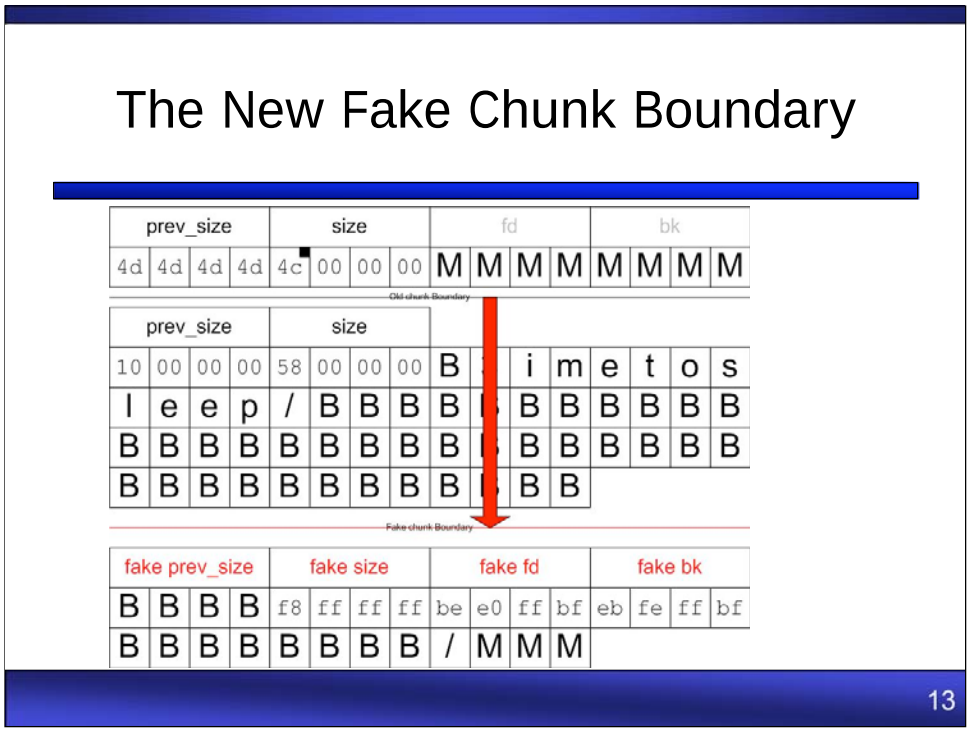
The least significant bit is '1', meaning the previous chunk is marked as allocated (previous chunk 'in use'). To calculate our actual SIZE, treat the least significant bit as a zero:

01001100 == "L" == 0x4c == 76



What does the addition of 60 bytes to the chunk's size accomplish? When the chunk is free()ed, the next chunk will be consolidated with the current chunk via forward consolidation. The next chunk's location is determined by adding the SIZE of the current chunk to the current chunk's address.

SIZE is now 76 bytes, so the chunk boundary is moved 60 bytes into the middle of the next chunk.



60 bytes lines up perfectly with the embedded fake heap structure.

In the fake chunk header:

•PREV_SIZE is BBBB (not important for the attack)

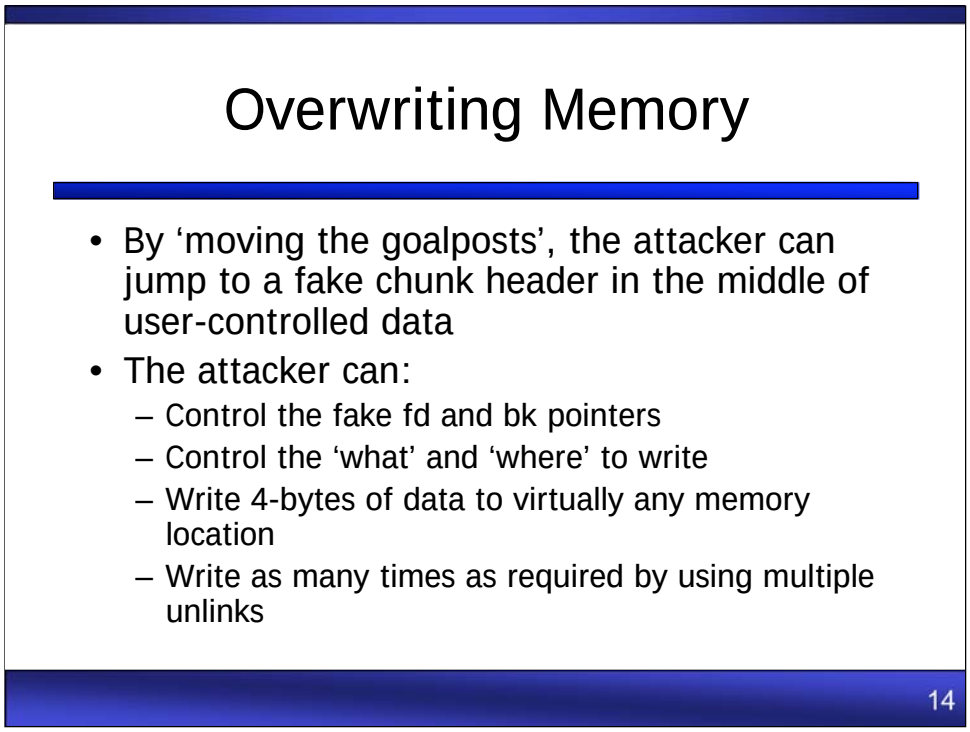
•SIZE is fffffff8 (-8), 8 bytes before this chunk, an area controlled by the attacker

•fd is bfffe0be

•bk is bffffeeb

As we will see shortly, 0xeb is the first byte of shellcode, which will be copied to location 0xbfffe0be.

The new SIZE is used for consolidation, and must be an even number so that the PREV_INUSE flag (least significant bit) is 0 (unallocated).



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`
  - `bfffffe15` is copied to location `bfffe0bf`
  - `bfffffe42` is copied to location `bfffe0c0`
  - `bfffffe4c` 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.

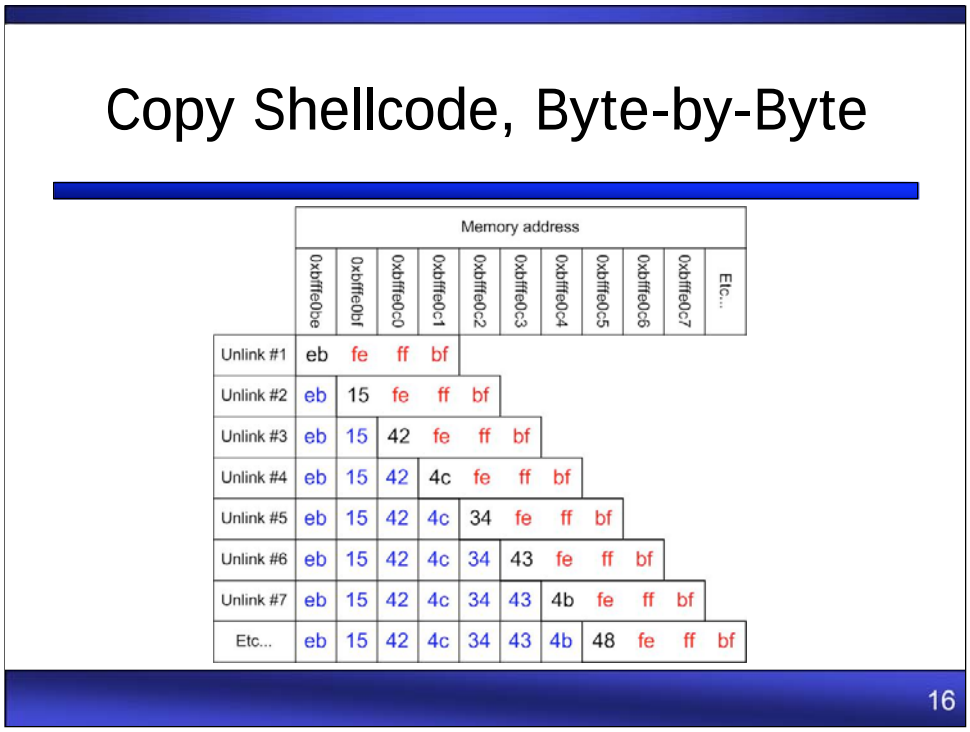
Unlink will copy each fake bk to the memory referenced in the fake fd field, in the order the chunks are unlinked.

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:

- `bfffffebe` is copied to location `bfffe0eb`
- `bfffe0bf` is copied to location `bfffffe15`
- `bfffffec0` is copied to location `bfffe042`
- `bfffffec1` is copied to location `bfffe04c`

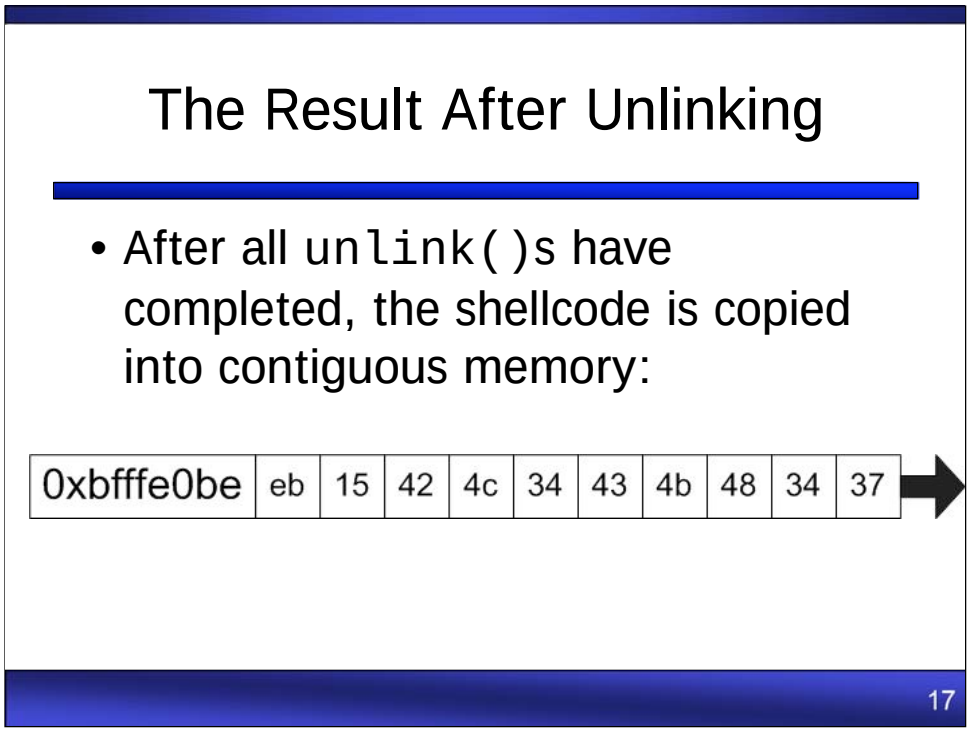
This damages a portion of memory, but has no effect on the attack.



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"

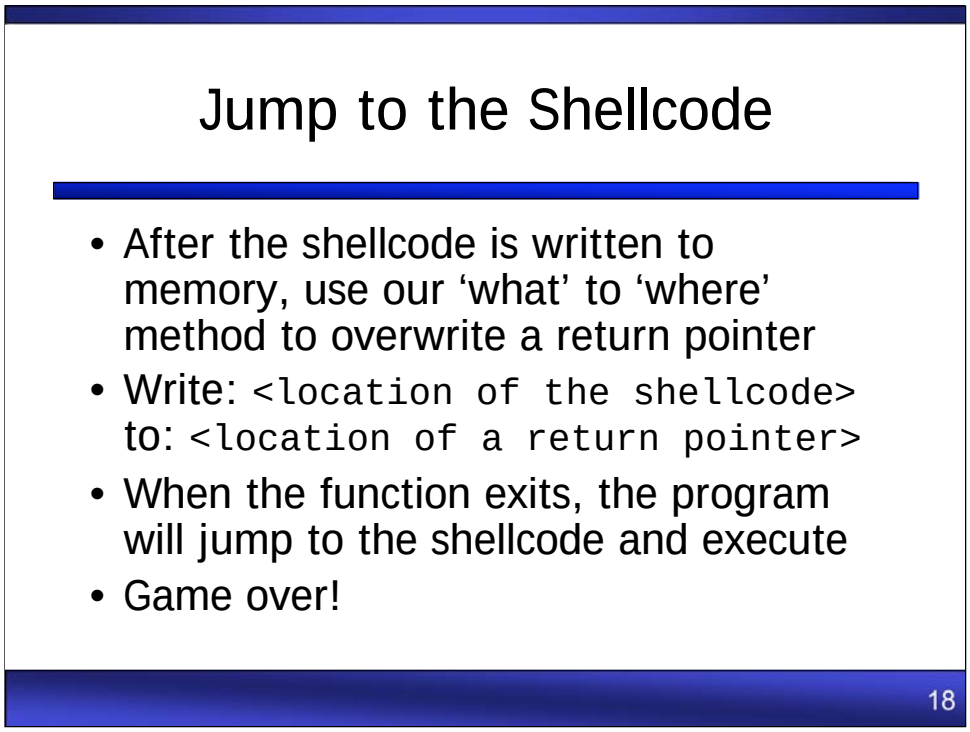


After unlink is complete, the shellcode is copied to memory.

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…).



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:

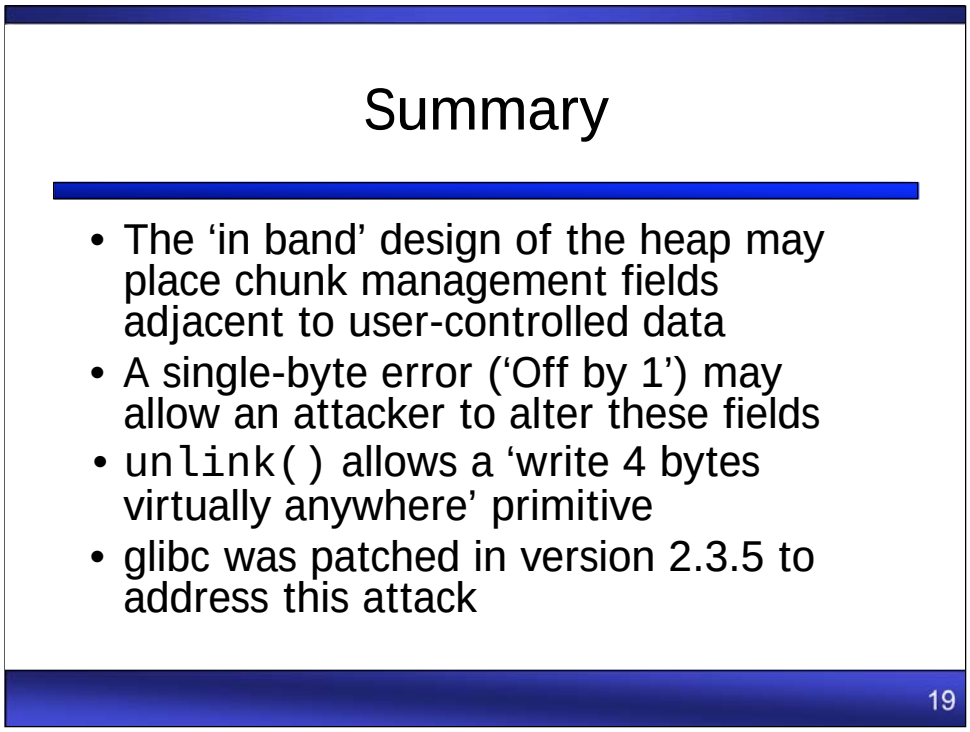
•The fake heap chunk headers will be unlink()ed

•‘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



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

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