Heap ‘Off by 1’ Overflow Illustrated

Eric Conrad
October 2007

The Attack

- Older CVS versions are vulnerable to an ‘Off by 1’ attack, where an attacker may insert one additional character into the heap
- CVS (Content Versioning System) is an open source version control system.
- This attack hinges on a single unaccounted ‘M’
- This attack illustrates critical techniques for maneuvering in memory

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

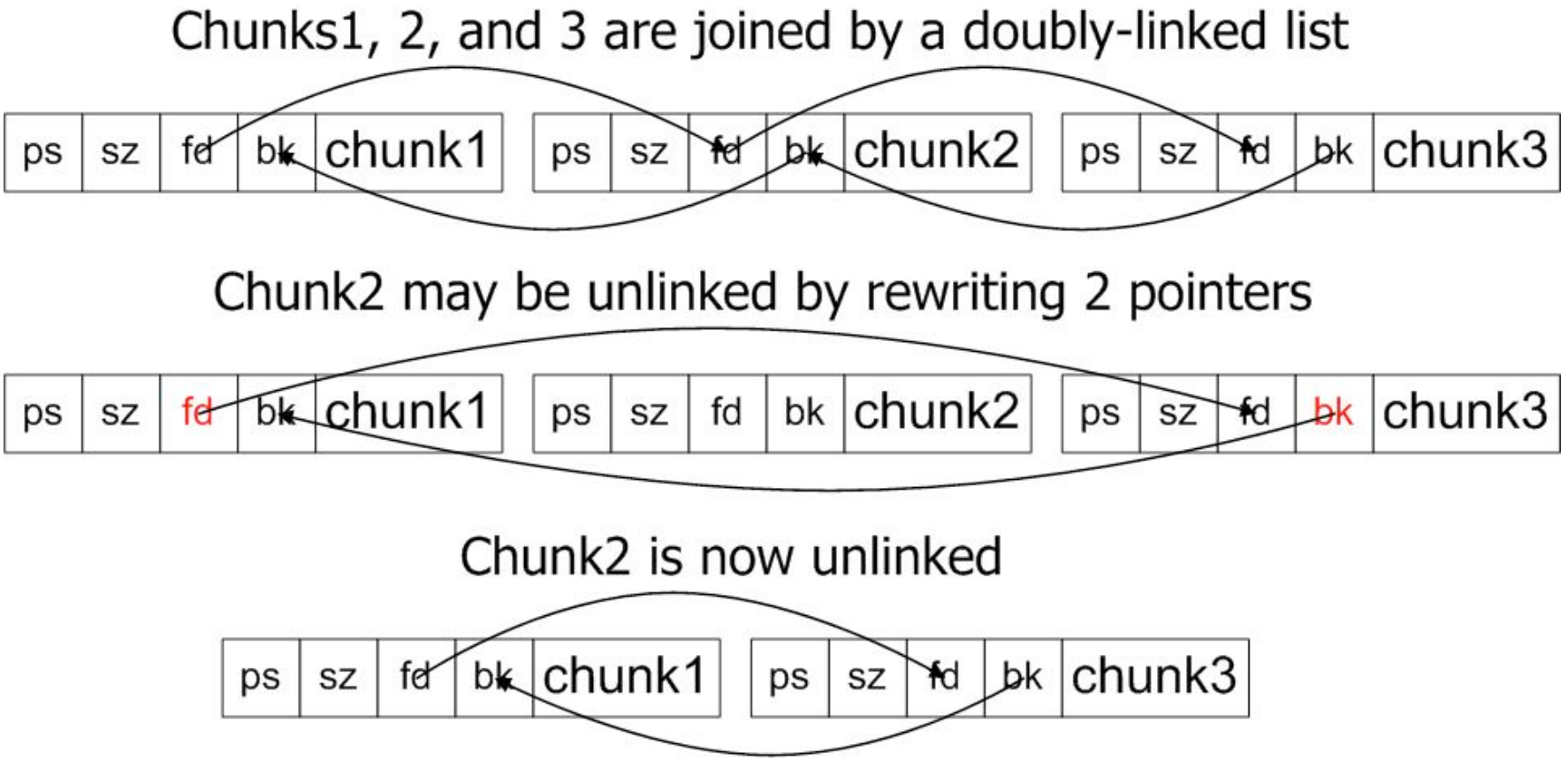
‘Off by 1’ Attack

- The CVS ‘Is Modified’ command appends an ‘M’ flag to the end of a CVS entry string
- A single call to ‘Is Modified’ appends one ‘M’ to the end of the CVS entry
- Due to a programming error, an attacker may call ‘Is Modified’ repeatedly, adding additional ‘M’s
- This is an ‘Off by 1’ attack

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

Unlinking a Chunk

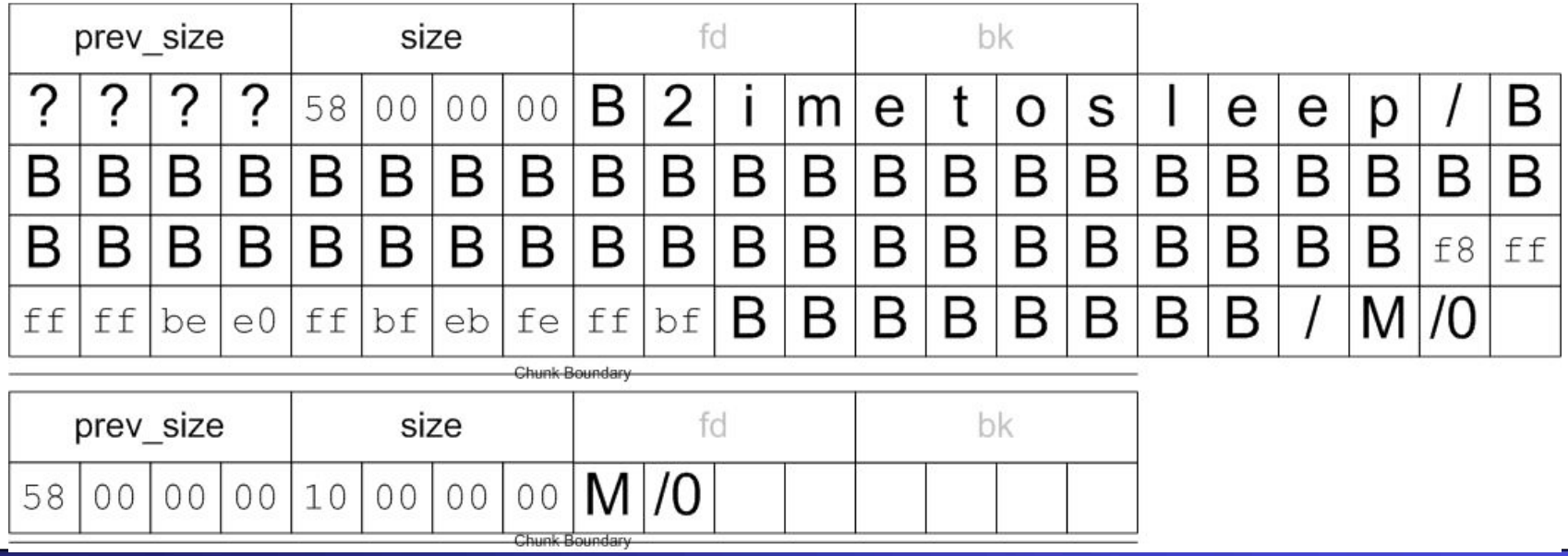


Unlink in more detail

- 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!

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

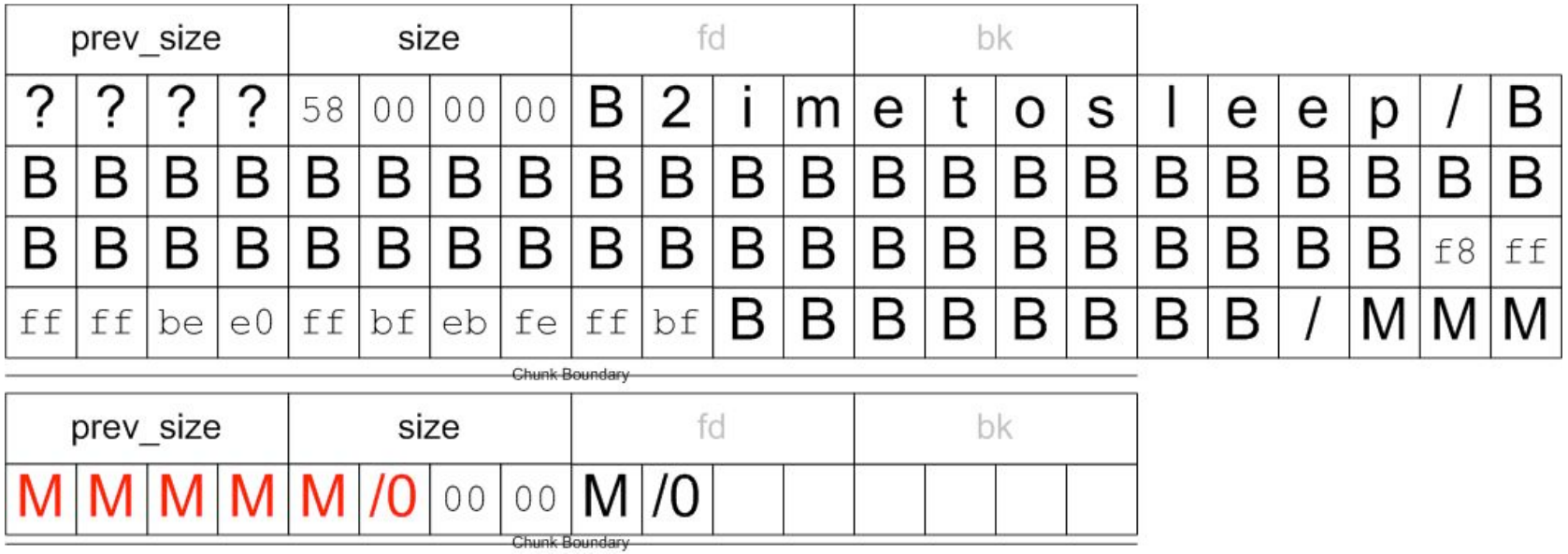


MMMMMMMM...

- Every call is ‘Is Modified’ will add an ‘M’ to the first chunk
- Eventually the ‘M’s will write past the chunk boundary, into the next chunk
- The next chunk’s PREV_SIZE and SIZE will change

The Heap After the Attack

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



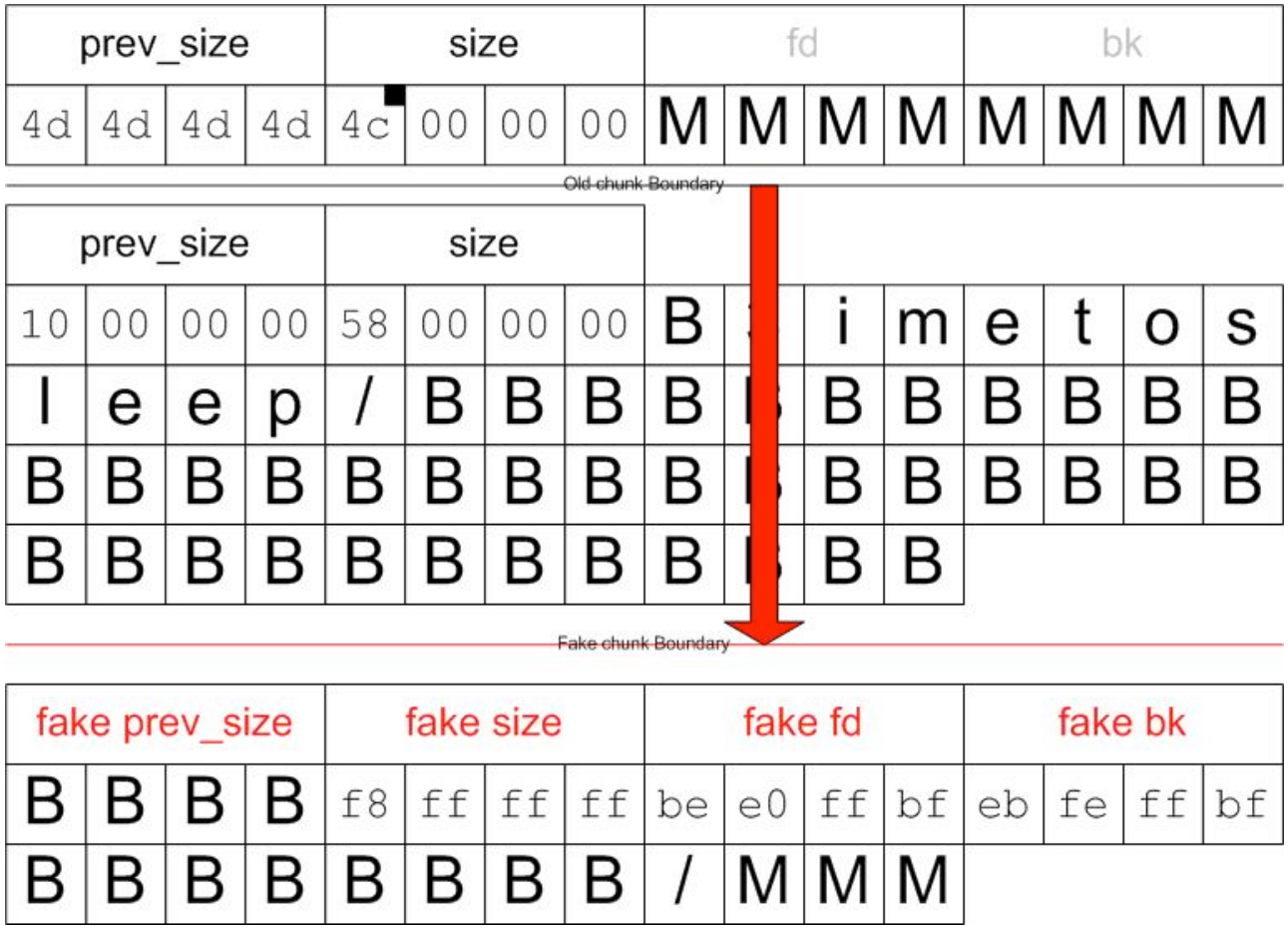
New SIZE

- The next chunk's PREV_SIZE has been changed to 0x4d4d4d4d ('MMMM')
  - This value is not important for the attack
- The next chunk's SIZE has been changed to 0x4d ('M')
  - This is the critical change!
- The next chunk's SIZE changed from 16 bytes to 76 bytes (+60 bytes)

Why Change SIZE?

- Unallocated chunks are consolidated and returned to available memory via `free()`
- Consolidation ‘jumps’ to the next free chunk, based on the current chunk’s SIZE
- Adding 60 bytes to SIZE makes it jump 60 bytes into the middle of the next chunk’s data
- The attacker controls the next chunk’s data!

The New Fake Chunk Boundary



Overwriting Memory

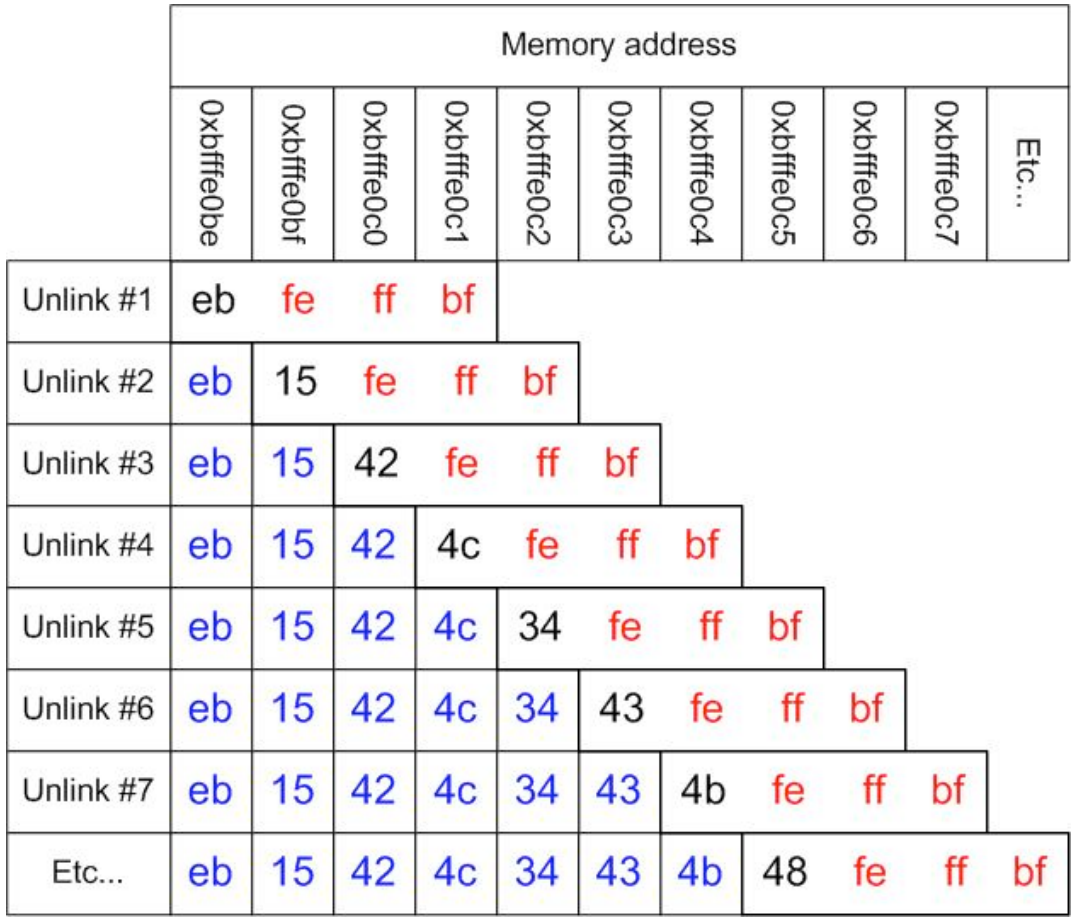
- By ‘moving the goalposts’, the attacker can jump to a fake chunk header in the middle of user-controlled data
- The attacker can:
  - Control the fake fd and bk pointers
  - Control the ‘what’ and ‘where’ to write
  - Write 4-bytes of data to virtually any memory location
  - Write as many times as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

Writing ‘what’ to ‘where’

- When the fake chunks are freed, unlink copies bk to location fd
  - `bffffeeb` is copied to location `bfffe0be`
  - `bfffe15` is copied to location `bfffe0bf`
  - `bfffe42` is copied to location `bfffe0c0`
  - `bfffe4c` 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	.	.

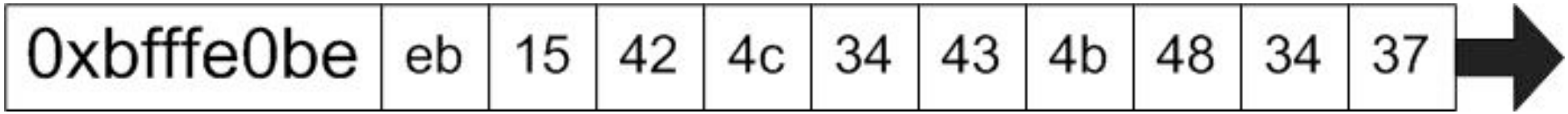
Copy Shellcode, Byte-by-Byte



	Memory address										
	0xbfffe0be	0xbfffe0bf	0xbfffe0c0	0xbfffe0c1	0xbfffe0c2	0xbfffe0c3	0xbfffe0c4	0xbfffe0c5	0xbfffe0c6	0xbfffe0c7	Etc...
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	bf

The Result After Unlinking

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



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!

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