A Heap of Trouble
Exploiting the Linux Kernel
SLOB Allocator

Dan Rosenberg
Who am I?

- Security consultant and vulnerability researcher at VSR in Boston
  - App/net pentesting, code review, etc.
  - Published some bugs
  - Focus on Linux kernel
  - Bad habit of rooting Android phones
  - Research on kernel exploitation and mitigation
Agenda

▪ What is SLOB?
▪ How does SLOB work?
▪ Evaluating exploitability
▪ SLOB exploitation techniques
▪ Demo
▪ Conclusion
Intro to SLOB
What is SLOB?

- Linux kernel supports three heap allocators:
  - SLAB, SLUB, and SLOB
  - Service dynamic allocations for kernel

- Implement `kmalloc()` and `kfree()` interfaces

- Sits on top of page frame allocator
Where is SLOB Used?

- Primarily embedded systems (low memory footprint)
  - Embedded Gentoo
  - OpenEmbedded
  - OpenWrt
  - Commercial embedded devices

- Mobile?
  - Not yet, maybe soon
Why is SLOB Interesting?

- Different allocation behavior and metadata from SLAB/SLUB
- No existing work on SLOB
- Who doesn't like crushing weak heaps?
Where Can I Use These Techniques?

- CVE-2009-1046: off-by-two heap overflow

- CVE-2010-2959: integer overflow leading to heap overflow in Controller Area Network (CAN)

- CVE-2010-3874: heap overflow on 64-bit platforms in Controller Area Network (CAN)
  - Not exploitable on any allocator but SLOB :-)

- CVE-2011-0699: heap overflow in btrfs

-CVE-2012-0038: heap overflow in XFS
How Does SLOB Work?

- Three singly-linked lists of partially-full pages
  - Less than 256 bytes
  - Less than 1024 bytes
  - Less than 4096 bytes

- Multiple sizes within same page

- `slob_page` struct
  - Metadata at base of actual SLOB page
  - Free units
  - Pointer to first free chunk within page
  - Linked list of free pages
Blocks

- Pages are broken into blocks (chunks)
- Size measured in SLOB_UNITS (2 bytes)
- Initially, each page is one big block
- Fragmented as necessary
SLOB Partially-Free Page Lists

free_slob_small

free_slob_medium

free_slob_large
Metadata

▪ Allocated blocks have 4-byte size header

▪ Free blocks have packed header
  ▫ If first two bytes are positive:
    ● First two bytes are size
    ● Second two bytes are index (in SLOB_UNITs) from base of page to next free block
  
  ▫ If first two bytes are negative:
    ● First two bytes are negative index to next free block
    ● Total size (including header) is assumed to be two SLOB_UNITs
Metadata Example

Size = 2  Next Free = 9  Free  Size = 2  Allocated

1  2  3  4  5  6  7  8  9
Allocation

- Choose appropriate linked list for size

- Walk list until page reporting enough room
  - Not guaranteed, could be non-contiguous
  - If no sufficiently free pages, allocate new page
Allocation

▪ Attempt allocation of size + 4 bytes (room for header)
  ▫ Walk free chunks checking sizes
  ▫ If exact fit, unlink
  ▫ If too big, fragment and unlink
  ▫ On failure, continue to next page

▪ Insert size metadata, return chunk

▪ Rotate linked list of pages
  ▫ Most recently used page is checked first
Freeing

▪ Freelist maintains address order

▪ Find freelist head for chunk (apply page mask to chunk address)

▪ Walk freelist until insertion point (address order)

▪ Adjust freelist metadata
  ▫ Prev->next => Chunk
  ▫ Chunk->size => size
  ▫ Chunk->next => Next
Evaluating Exploitability
Exploitability Criteria

▪ What makes a heap “exploitable”?

▪ Criteria would be useful in evaluating heaps besides SLOB

▪ Can compare different heap implementations
Allocation behavior

“To what degree can attackers predict and control locality of allocations and frees?”
Allocation Behavior in SLOB

▪ No randomness in allocations

▪ Once a fresh page is allocated, all allocations are guaranteed to be consecutive within page

▪ Objects are freed predictably
  ▫ Inserted into list in address order
Object Co-Residency

“Do multiple types of objects exist in the same memory region?”
Object Co-Residency in SLOB

- Unlike SLAB/SLUB, all objects share same cache
- Size is only factor in determining where to allocate
- Unlike SLUB, no per-cpu caches
Object Metadata

“Do free or allocated objects contain inline metadata that can be exploited?”
Object Metadata in SLOB

- SLAB/SLUB have minimal inline metadata (next free pointer), but SLOB has:
  - Allocated chunk size field
  - Free chunk size field
  - Free chunk list index field
Exploitation Mitigation

“Are any hardening measures in place to deter exploitation of heap vulnerabilities?”
Exploitation Mitigation in SLOB

NOTHING TO SEE HERE

KEEP WALKING
# Heap Comparison

<table>
<thead>
<tr>
<th>Allocation Behavior</th>
<th>SLOB</th>
<th>SLUB</th>
<th>Windows 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Co-Residency</td>
<td>Moderate</td>
<td>Easy</td>
<td>Moderate</td>
</tr>
<tr>
<td>Object Metadata</td>
<td>Moderate</td>
<td>Easy</td>
<td>Moderate</td>
</tr>
<tr>
<td>Exploitation Mitigation</td>
<td>Moderate</td>
<td>Easy</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

**Exploit Difficulty**
- Easy
- Moderate
- Difficult
Pre-Exploitation
Goals of Pre-Exploitation

▪ Cause heap to be in state conducive to exploitation

▪ Requires knowledge of allocation behavior

▪ Usually requires knowledge of specific allocation primitives
  ▫ Can trigger allocation and/or freeing of objects of specific sizes
Pre-Exploitation on SLOB

- In classic heap overflow, goal is usually adjacent blocks

- In SLOB, once fresh page is used, allocations will be contiguous (for the short term)

- Basic approach:
  - Find allocation primitive for appropriate list size
  - Trigger enough allocations to cause fresh page
  - Trigger allocations and frees to cause vulnerable object to be placed appropriately
How Much Should I Allocate?

- No /proc/slabinfo on SLOB

- Have to make a reasonable guess
  - Depends on system uptime and load
  - No real penalty for allocating too much

- Experimentally, a few hundred allocations is plenty
Pre-Exploitation on SLOB

- Rotation of partially-free page list is helpful
  - Can fill partially free pages with larger objects
  - Subsequent smaller allocations will be in fresh page, even though they might have fit in other partially full pages
Exploitation
Assumptions

- We have some heap overflow vulnerability
  - Can write data past the end of a heap chunk into the next chunk

- Degree of control over length and contents will vary

- Can find appropriate allocation primitives
  - Structures with function pointers, etc.
Arbitrary Overflow

Full control over size of overflow and contents
Object Data Overwrite

- Fill partial pages and cause allocation of fresh page
  - We'll assume this from now on...

- Position target chunk after vulnerable chunk

- Trigger overflow

- Trigger function pointer call/write to pointer
Object Data Overflow Cleanup

- Unlike SLUB/SLAB, allocated chunks have 4-byte size header
  - Need to restore to avoid unwanted corruption

- If new size is less than old size, do nothing
  - No freelist corruption, shrinking causes no harm

- Otherwise, cleanup after gaining control
  - If function pointer call, base of chunk is almost guaranteed to be in a register
Off-by-Small Overflow

Some control over contents of three to four byte overflow
Free Pointer Overwrite Overview

- Modification of technique by sgrakkyu and twiz

- Basic approach: corrupt freelist to trigger chunk reuse
  - If we can trigger allocation of a useful target block on top of data we control (or vice versa) we can win

- Need to corrupt “next free” pointer in adjacent free block

- Remember: it's a two-byte index, not a pointer
Free Pointer Overwrite #1

▪ Do the pre-exploitation dance

▪ Fill fresh page with target chunks

▪ Trigger overflow into free chunk, overwriting 3-4 bytes (size and one or two bytes of next free pointer)

▪ Trigger allocation of controlled chunk on top of some target block

▪ Win
Free Pointer Overwrite #1

slob_page  fptrs    fptrs    vuln    target   (free)   (free)

slob_page  fptrs    fptrs    overflow  target   (free)   (free)

slob_page  fptrs    fptrs    overflow   (alloc)   (free)   (free)

slob_page  fptrs    fptrs    overflow   (alloc)   (free)   (free)
Free Pointer Overwrite Cleanup

- Freelist has been corrupted
  - Subsequent allocations may panic the kernel

- Easiest option is to terminate the freelist early (thanks Nico) so corrupted free chunks never get traversed
  - Chunk is considered “final” when its next-free index returns a next chunk that is page aligned
  - Overwrite a free chunk's next pointer with NULL or any multiple of 0x800 to terminate the list
Off-by-Smaller Overflow

Some control over contents of one to two byte overflow
Free Pointer Overwrite #2

- Same as other free pointer overwrite, except:
  - Take advantage of special case
  - Negative value in first two bytes of free chunk is interpreted as negative index, not size

- Allows exploitation of controlled off-by-two overflow (need both bytes to overwrite with negative two-byte value)

- Remember to clean up the freelist
Off-by-One Overflow

Some control over contents of one byte overflow
Chunk Growth Attack

▪ Overwrite size field on adjacent free or allocated chunk to “grow” that chunk
  ▫ Shrinking does nothing useful – no freelist corruption, so just causes wastage of memory

▪ If overflow into allocated block, cause that block to be freed

▪ Trigger allocation of chunk with size equal to “grown” size with data you control

▪ Second portion of this chunk will overlap with target chunk, allowing exploitation
Chunk Growth Attack

slob_page (alloc) vuln target fptrs (free)

slob_page (alloc) overflow target fptrs (free)

slob_page (alloc) overflow target fptrs (free)
Off-by-One NULL Byte Overflow

Well, this sucks.
What Are Our Options?

▪ Allocated chunk size header
  ▫ NULL byte means we can only shrink, not useful

▪ Free chunk size header
  ▫ Same as above

▪ What was that special case again?
Special Case

▪ If first two bytes are negative:
  ▫ Size is assumed to be one SLOB_UNIT (2 bytes)
  ▫ First two bytes are negative index to next free block

▪ Great, overwriting LSB of free index could be a win
  ▫ Trigger allocation on top of existing chunk

▪ All we need to do is cause a 2-byte block to be allocated!

▪ But...
mm/slob.c:

void *__kmalloc_node(size_t size, gfp_t gfp, int node)
{
  ...
  int align = max(ARCH_KMALLOC_MINALIGN, ARCH_SLAB_MINALIGN);
  ...
  m = slob_alloc(size + align, gfp, align, node);
  ...
}

include/linux/slab.h:

#define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
What Does This Mean?

▪ The only piece of metadata we can possibly exploit can't exist in any chunks we can allocate :-(

▪ Is all hope lost?
  ▫ Hint: no.

▪ Remember how SLOB works: chunks of varying sizes exist in the same cache
Fragmentation to the Rescue!

▪ Same old pre-exploitation phase, fill new page with targets

▪ Trigger allocation and freeing of block four bytes larger than size of vulnerable block

▪ Trigger allocation of vulnerable block

▪ SLOB will fragment previous block into vulnerable block and four-byte “special” chunk

▪ Trigger overflow, continue as if free pointer overwrite
Fragmentation Attack: Phase 1

- slob_page
  - alloc
  - free

- slob_page
  - alloc
dummy
  - alloc
  - free
  - fptrs
  - fptrs

- slob_page
  - alloc
  - free
  - alloc
  - free
  - fptrs
  - fptrs

- slob_page
  - alloc
  - vuln
  - alloc
  - free
  - fptrs
  - fptrs
Fragmentation Attack: Phase 2

slob_page (alloc) vuln (alloc) (free) fptrs fptrs

slob_page (alloc) overflow (alloc) (free) fptrs fptrs

slob_page (alloc) overflow (alloc) (free) fptrs fptrs
Demo
Setting up a Test Environment

▪ Wrote LKM “playground”

▪ Creates device file

▪ Can trigger heap primitives via ioctl
  ▫ Allocate, free, overflow, function pointer call, etc.

▪ Develop techniques with theoretical primitives
  ▫ Replace with real examples later
Chunk Growth Attack Demo
Conclusion

▪ SLOB's design allows easy exploitation

▪ SLOB has virtually no hardening
  ▫ Basic freelist validation would be simple
    • Next chunk is after current chunk
    • Next chunk is before end of page

▪ See KERNHEAP for ideas
Future Work

▪ Harden the SLOB allocator?
  ▫ I'm not going to do this

▪ Automated finding of heap primitives
  ▫ I don't know anything about static analysis
    • Need to trace code paths, enumerate all heap activity, and determine which chunks remain allocated persistently
  ▫ Jon Oberheide's kstructhunter is a start
Thanks To...

- twiz
- Tarjei Mandt
- Nico Waisman
Questions?

E-mail: drosenberg@vsecurity.com
Twitter: @djrbliss

Company:
http://www.vsecurity.com

Personal:
http://www.vulnfactory.org