Large memory management vulnerabilities

System, compiler, and application issues

Gaël Delalleau

gael.delalleau@beijaflore.com
gael.delalleau+csw@m4x.org

Security consultant from

http://www.beijaflore.com

CancSecWest 2005
Vancouver – May 4-6
Agenda

- Dynamic map of a process virtual memory space

  - Operating systems & compilers security issues
  - Exploiting unexploitable bugs
  - Application flaws dealing with large data sizes

- Easy to exploit? Easy to protect from?
Introduction (1/2)

- Large (N GB) memory sizes now common
  - Memory size = RAM + swap
  - Servers
  - Desktop (games, multiple apps)

- A process can alloc 1 to 3 GB depending on OS
  - An application may need to handle large (N GB) data sizes
  - 32 bits CPUs: the whole virtual memory space can be filled
  - “Out of memory” situations (OOM) in a process are not fatal

- 64 bits CPUs give much more virtual space
Introduction (2/2)

Big memory usage situations are badly handled, introducing exploitable holes in applications

- Operating systems: break the usual behavior rules about stack, heap, mappings at page 0...
- Compilers: introduce security flaws in valid application code
- Applications: 32 bits counters overflow and sign problems

“Unexploitable” bugs may be exploited to run arbitrary code in a process

- NULL pointers dereference, common in OOM conditions
- Buffer overflows and underflows may corrupt an adjacent memory area
Dynamic map of a process virtual memory space
Dynamic map of a process virtual memory space

- Naive view
- Naive view memory protections
- Solaris 10
- FreeBSD 5.3
- Linux 2.6
Virtual memory space: naive view

- Protected memory reserved for the OS (kernel)
- Memory holding environment, arguments, local variables and execution flow control structures: “stack”
- Dynamically allocated data: “heap”
- Static data
- Executable code

UNMAPPED MEMORY

Executable code

Static data

Dynamically allocated data: “heap”

Memory holding environment, arguments, local variables and execution flow control structures: “stack”

Protected memory reserved for the OS (kernel)

Large memory management vulnerabilities
Memory protections (naive)

 Protected memory reserved for the OS (kernel)

 Memory holding environment, arguments, local variables and execution flow control structures: “stack”

 A “guard page” prevents heap and stack from colliding

 Dynamically allocated data: “heap”

 Static data

 Executable code

 Trying to access unmapped memory throws an exception or a segmentation fault signal
 => protects against NULL pointer dereference
Let's dive into the real world

- **No standard** for memory allocation behavior

- **Major changes between vendors and versions of:**
  - OS
  - Libc
  - Threading library
  - Compiler and linker

- **Additional mappings**
  - Dynamic libraries: code and data
  - Additional heaps and stacks (threads...)
  - Anonymous memory (mmap, VirtualAlloc...)
  - Shared memory (IPC)
  - Files mapped in memory
  - System mappings: PEB, TEB (Windows), vsyscall (Linux), ...

- **Next slides show the behavior of real systems**
Solaris 10 / x86

Status of unallocated memory:

- Mapping forbidden
- Mapping allowed
- Mapping allowed, but can't be reached with default limits
- Gap area, mapping impossible

Mapping allowed, but can't be reached with default limits

“top down” mmap area [fragmented]

Heap growing up [continuum]

Stack growing down [continuum]

Upper and lower limits of the heap
Large memory management vulnerabilities

Linux 2.6

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed

Mapping allowed:
- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]

Mapping forbidden:
- "top down" mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]

Lower limit of the stack (default 128 M)
Memory management vulnerabilities

**FreeBSD 5.3**

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- Mmap area “from bottom to top” [fragmented]
- Heap growing up [continuum]
- Stack growing down [continuum]
- Limit between heap and mmap area

**Status of unallocated memory**:  
- Mapping forbidden
- Mapping allowed
Operating systems and compilers security issues
Operating systems and compilers security issues

- Heap / stack overlap
- Jumping the stack gap
- Example of a memory management kernel bug
Can heap and stack “collide”?  
- Heap grows up... stack grows down...  
- Collision or not: depends on process VM map  
- Two protections mechanisms at bottom of stack
  - Gap page(s): mappings forbidden  
  - Guard page(s): PROT_NONE mapping  

Linux 2.6
- No gap, no guard page!  
- mmap() allocates close to bottom of stack if low mem (kernel>=2.6.9 ?)  
- Heap allocations use mmap if size>128K or if low memory condition  
- Thus heap and stack can be contiguous!
Large memory management vulnerabilities

**Linux 2.6**

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- "Top down" mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]
- Lower limit of the stack (default 128 M)

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed

Memory management vulnerabilities
Large memory management vulnerabilities

Linux 2.6

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- "top down" mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]
- Lower limit of the stack (default 128 M)

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed
Large memory management vulnerabilities

**Linux 2.6**

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- “top down” mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]
- Lower limit of the stack (default 128 M)

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed

The diagram illustrates the memory map of a Linux 2.6 system, showing the allocation of memory regions such as ELF (Executable and Linkable Format), stack, heap, unmapped memory, and library memory (libs). The unallocated memory is indicated by different colors and patterns, with mapping allowed and forbidden areas visually represented.
Linux 2.6

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- "top down" mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]
- Lower limit of the stack (default 128 M)

Memory management vulnerabilities
**Large memory management vulnerabilities**

**Linux 2.6**

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- "top down" mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]

**Memory and stack**

- Heap and stack are contiguous.
- There is no unmapped memory at bottom of stack.
- Stack growth and stack overflow signaling are no longer handled by the kernel since they rely on page faults access at bottom of stack.
Large memory management vulnerabilities

Linux 2.6

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- “top down” mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]
- Lower limit of the stack (default 128 M)

Stack growth needed by application

%esp lies in heap mapping
(for purists: the stack and heap VMAs don't overlap)
Heap / stack overlap Demo

Exploiting mod_php 4.3.0 on Apache 2.0.53

► Goal: execute assembly code from a restricted PHP script
► Allows for breaking out of safe_mode
► Needs ability to allocate ~3 GB of memory
  - Enough RAM + swap
  - Disabled PHP memory_limit option, or use a memory leak

Exploit scenario

► Allocate large blocks of memory with emalloc() => malloc()
► Call recursive function many times
  - the stack “goes down” and overlap with one of the allocated block
  - R/W access to this block == R/W access to stack memory :-)
  - Modify a saved EIP address in stack to point to shellcode and return
Vulnerability Status
for heap / stack overlap

- Linux 2.4: SAFE
- Linux 2.6: UNSAFE
- FreeBSD 5.3: MMAP UNSAFE
- OpenBSD 3.6: SAFE (but...)
- Linux emulation on FreeBSD 5.3: UNSAFE
- Linux emulation on OpenBSD 3.6: SAFE (but...)
- Solaris 10 / x86: SAFE
- Solaris 9 / Sparc: SAFE
- Windows XP SP1: SAFE
- Any OS with certain uncommon threading libraries: UNSAFE
Jumping the stack gap

- Protection with gap or guard page: unsafe
  - A few KB under the stack are protected by the OS
    - No other mapping can lie there
    - OS grows the stack mapping if a GP fault happens below the stack
    - If the stack can't be grown a SIGSEGV is delivered
  - BUT: the *application* controls the stack pointer, not the OS
    - Local ("automatic") variables allocation on function calls
    - Usage of alloca()

- Vulnerability is not in the application C code...
- ... but may be introduced by the *compiler*
Normal behavior

Stack (MAP_GROWSDOWN mapping on Linux)

Gap or guard page(s) : a few KB

%esp

%esp - 10008

int
f(char *src) {
    char buffer1[5000];
    char buffer2[5000];
    memcpy(buffer2, src, 5000);
    ...
    return 0;
}

Other mappings
Normal behavior

Stack (MAP_GROWSDOWN mapping on Linux)

Gap or guard page(s) : a few KB

%esp - 10008

int f(char *src) {
    char buffer1[5000];
    char buffer2[5000];
    memcpy(buffer2, src, 5000);
    ...
    return 0;
}

%esp

Other mappings
Jumping the gap

Stack

Gap or guard page(s): a few KB

Heap or other interesting mappings allocated just under the stack

```
int f(char *src) {
    char buffer1[5000];
    char buffer2[5000];
    memcpy(buffer2, src, 5000);
    ...
    return 0;
}
```
Jumping the gap

Stack

%esp

Gap or guard page(s) : a few KB

%esp - 10008

Heap or other interesting mappings allocated just under the stack

int f(char *src) {
    char buffer1[5000];
    char buffer2[5000];
    memcpy(buffer2, src, 5000);
    ...
    return 0;
}

Heap or other interesting mappings

OVERLAP

with stack!
GCC default behaviour unsafe

Allocation of local variables on stack

Prologue: sub $size, %esp

- $size > size of protected gap \( \geq \) size of a page (4096 B)
- Area not accessed until told by the application
- If a mapping exists at \%esp = \%esp\_old - \$size, then access to \%esp doesn't create a GP fault: “jumping the gap”

Same problem with alloca(size)

- Inlined as sub $size, %esp
- No sanity check on $size
  - $size can be larger than gap size
  - $size can be negative
Example on Solaris 9 / Sparc

- Big gap, between 16 KB and 64 KB
  - depends on stack size limit

- ld.so is mapped below the stack size limit
  - If we jump the gap, stack variables will overwrite the .data and .bss sections of ld.so

- Most applications will not be vulnerable
  - Need for a function with a huge unused local variable or alloca()
  - We must access this function when %esp is close from stack limit

- We can control %pc on vulnerable applications
  - ld.so data section has pointers to function pointers, which are called
Forcing safer stack allocations

- Use gcc flag `-fstack-check`
  - A NULL byte is written every 4096 bytes in the allocated area
  - The gap or guard page will be hit, forcing stack growth
  - If stack is unable to grow the kernel delivers a signal to the process
  - *This is the default on Windows*
    - the kernel uses only guard page accesses to grow the stack
    - access below the guard page would trigger a 0xC0000005 exception

- `alloca(size)` also checked...

- ...yet negative sizes (>2G) are still unsafe
Vulnerability Status for “gap jump”

- GCC on UNIX (default)          UNSAFE
- GCC on UNIX (with -fstack-check) SAFE
- Other compilers on UNIX         UNTESTED
- Any good compiler on Windows    SAFE
Example of a memory management bug in kernel

Small flaw in mmap() allocations on OpenBSD

- size = 3,9G => error, mmap() returns -1 (OOM)
- 4G–4096 < size <= 4G => success, but allocates nothing

The flaw lies in kernel code

```c
#define round_page(x)   (((x) + PAGE_MASK) & ~PAGE_MASK)

In sys_mmap(p, v, retval):
    pageoff = (pos & PAGE_MASK);
    size += pageoff; /* add offset */
    size = (vsize_t) round_page(size); /* round up */
    if ((ssize_t) size < 0) return (EINVAL); /* don't allow wrap */
```

Some applications might be at risk

- If mmap() call with a size parameter we can control (file mapping?)
- Exploitation: access to other mappings instead of the expected one
Exploiting unexploitable bugs
Exploiting unexploitable bugs

- Exploiting NULL pointers (OOM crashes)
- Exploiting other bugs using mapping overflows
Exploiting NULL pointers

Using OOM “crashes” to run arbitrary code

- `malloc(size)` returns NULL (00000000) if OOM
  - Flawed applications fail to check this return value
  - Dereferencing the NULL pointer access unallocated memory => OS sends SIGSEGV or exception
  - This is the expected behavior (documented)... BUT!

- We might be able to exploit this to run ASM code
  - On some OSes we can map the first page at address 0
  - On some applications the address really accessed is not 0

Exploiting more bugs
Creating a mapping at address 0

- On Linux 2.6.x `mmap()` can allocate the first page
  - So `malloc()` can too
  - We just need to fill the available memory space

- On Solaris 10/x86, the stack can “grow” down to 0
  - But only if the default stack size limit has been increased
Large memory management vulnerabilities

Solaris 10 / x86

ELF mapping: code segment [r-x]
ELF mapping: data segment [rwx]
“top down” mmap area [fragmented]
Heap growing up [continuum]
Stack growing down [continuum]
Upper and lower limits of the heap

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed
- Mapping allowed, but can't be reached with default limits
- Gap area, mapping impossible

Mapping allowed, but can't be reached with default limits
Linux 2.6

- ELF mapping: code segment [r-x]
- ELF mapping: data segment [rwx]
- "top down" mmap area [fragmented]
- Heap growing up [fragmented]
- Stack growing down [continuum]
- Lower limit of the stack (default 128 M)

Status of unallocated memory:
- Mapping forbidden
- Mapping allowed
Example

Sample vulnerable code

/* Let's copy 'userdata' into 'buffer' */
size = strlen(userdata) + 1;
buffer = (char *) malloc(size); // no check of return value
memcpy(buffer, userdata, size); // buffer may be 00000000

On Linux 2.6  heap overflow situation

On Solaris 10/x86  stack overflow situation
Vulnerability Status

for memory allocation at 0

- Linux 2.4: SAFE
- Linux 2.6: UNSAFE
- FreeBSD 5.3: SAFE
- OpenBSD 3.6: SAFE
- Linux emulation on FreeBSD 5.3: SAFE
- Linux emulation on OpenBSD 3.6: SAFE
- Solaris 10 / x86: UNSAFE
- Solaris 9 / Sparc: SAFE
- Windows XP SP1: SAFE
Table offsets

Access to buffer[i] == *(buffer+i) instead of *buffer

- Means access to *(i) if buffer is NULL
- Can be in a valid mapping!
  - Depends on how much control we have on the index i
  - Depends on how close to address 0 we can put a mapping

Vulnerable code sample

```c
numberOfMessages += 1;
buffer = realloc(buffer, sizeof(imapFlags) * numberOfMessages);
(...)
// no check to see if buffer resizing has failed
buffer[numberOfMessages-1] = messageFlags;
// could be a write to *(0+numberOfMessages-1)
```
C++ “NULL” objects

- A high-level allocation function might return a “NULL” instance of a C++ class on failure
  - Static object stored in .(ro)data
  - Heap corruption may happen if return value is not checked

Vulnerable code real-life example (Mozilla)

When it fails to allocate memory, the `ReplacePrep` function "nullifies" the string:

```cpp
mData = NS_CONST_CAST(char_type*, char_traits::sEmptyBuffer);
mLength = 0;
```

But in `nsTSubstring_CharT::Replace` the return value was not checked:

```cpp
size_type length = tuple.Length();
cutStart = PR_MIN(cutStart, Length());
ReplacePrep(cutStart, cutLength, length);
if (length > 0)  tuple.WriteTo(mData + cutStart, length);
```

Exploiting more bugs
Exploiting other bugs with mapping overflows

- No gap / guard page enforced between mappings
  - Not enough protections on most systems
  - Contiguous mappings happen if large memory usage
  - Allows us to turn an overflow or underflow...
  - ... into corruption of another memory area

- May help to solve some exploitation problems
  - Difficult heap buffer overflows (end of heap, new GLibc and XP SP2 protections)
  - Large size memcpy heap overflows (-1 == 4G) that would trigger a crash
  - If a mapping is allowed on top of stack (threads, grsec random stack...):
    - Stack buffer overflow in argv() or env, in main() when main() never returns, big overflows with propolice-like protection...
  - Buffer underflows (also on stack on Linux 2.6)
Large memory management vulnerabilities

Example: exploiting Windows XP heap overflows with a mapping overflow

By filling the memory we may allocate heap next to Thread Environment Blocks (TEB)...

Overflow of controlled size

Heap

Large size overflow (memcpy with size -1...)
Example: exploiting Windows XP heap overflows with a mapping overflow

By filling the memory we may allocate heap next to Thread Environment Blocks (TEB)...

...and overwrite the SEH chain pointer of our thread's TEB to redirect execution flow

Sanity checks in ntdll.dll can be bypassed by tuning the “top of stack” and “bottom of stack” pointers.
Application flaws dealing with large data sizes
Application flaws dealing with large data sizes

- Analysis of integer overflows in 32 bits counters
- Handling of library calls in OOM situations
- The MMAP_FIXED aberration
32 bits integer overflows when handling large data

- Impossible to allocate 4G on 32 bits CPU
- Nevertheless math calculations can make integer wrap
  - Multiply by 3 (base64 decoding...): if we can allocate 1.3G contiguous
  
  ```c
  int len = strlen(data);
  bufLen = (len * 3)/4;
  ```
  - Multiply by 2 (string escape, buffer growth...): 2.7 G cont. max on Lin. 2.6

32 bits integer sign problems

- Appear when len > 2G
  ```c
  int len = strlen(ptr);
  if (len > buf_length) buf = realloc(buf, len+1);
  strcpy(buf, ptr);  // overflow if reallocation was not done
  ```
Handling of library calls in OOM situations

Idea: applications do not check return values for library calls that are trivial or that “always” success, but some of them will not do their job in OOM situations

- Applications make wrong assumptions about the actions they took
- It may create application malfunction and errors, some potentially exploitable

Needs more research!
Usage of MMAP_FIXED

- Calling mmap() with the MMAP_FIXED parameter destroys any previous mapping at the address.
- It is impossible to safely predict an address where a hole will exist in memory.
  
  ▶ Example for Linux programs:
  - Differences in allocation behavior between Linux 2.4 and 2.6
  - Linux emulation layers on other OSes has a different behavior
  - User interaction (big memory allocations)

- Thus, its use is unsafe.
Easy to exploit?
Easy to protect from?
Easy to exploit? Easy to protect from?

- System limitations
- Network limitations
- Protecting ourselves
System limitations

**Memory size**
- But VM can be overcommitted if it is not accessed

**Resource limitations**
- Stack size, data size, total VM size...

**Allocation speed**
- RAM is quick even for GBs
- Becomes quite slow (minutes) when switching to disk swap
Network limitations

Upload speed
- GBs can be sent in minutes or hours on modern LAN and Internet
- Services timeouts are a problem
- Data zip-bombs, memory leaks, multithreading... can help

Lack of information about the target
- Brute force is likely to be needed for these attacks
Protection

- GCC flag `-fstack-check`
- On Linux 2.4: increase `heap_stack_gap` in `/proc`
- Application code security audit
- Memory limits handled by the application (PHP...)
  - Memory leaks are to be treated as security bugs ;-) 
- Resource limitation enforced by the OS
  - Two edges sword
- Vendor patches?
“Don't panic”

- Very specific conditions may be necessary to succeed in real life with these techniques.
- Vulnerabilities introduced by OSes will be patched.
- Switching to 64 bits will solve some of these issues... and introduce new ones (ex: the `long` type is 64 bits on Unixes, but is 32 bits on Windows!)
- Yet some applications, on some systems, meet the conditions and critical exploitable vulnerabilities exist. (but don't panic and go audit your code)
Conclusion

- Usage of large quantities of memory in modern computing introduces unexpected vulnerabilities.

- Security holes may exist in applications even if their code is valid: the OS and the compiler break usual safe assumptions.

- Some of these flaws will be patched...

- ... others are classes of vulnerabilities that will last.

  ▶ More research needs to be done:

  - The area is broad, and my time is limited.
  - What about other OSes, threading libraries, compilers, embedded systems, emulation layers (Linux...), virtual machines...
  - How many applications are exploitable in real life situations?
Thank you!
Any questions?

Feedback, questions, comments... are welcome

gael.delalleau@beijaflore.com

gael.delalleau+csw@m4x.org

These updated slides will be available on

http://www.cppsecurity.com

Thanks to Solar Designer and H. D. Moore. All errors are mine.