# Heap memory safety

## Summary

In this exercise, we will learn about heap memory safety.

## Prerequisites

Setup an Ubuntu VM as outlined in the VM setup instructions on Blackboard.

## Details

### Part I-Heap memory overview

* Memory in the heap is managed by a dynamic memory allocator, or memory manager
* Memory managers organize memory using a construct called chunks
  + chunks are simply various sized blocks of n-contiguous bytes in memory
  + chunk sizes range from block-sizes with only a few bytes, to much larger blocks
* Memory managers oversee memory by
  + maintaining a list of available space (i.e., what is allocated, or in use, and what is free, or available)
  + finding and reserving chunks of contiguous bytes when requested
  + un-reserving chunks when they are no longer needed
* C standard defines an API for the standard memory management functions and these include
  + *malloc(size\_t size)*-allocates size bytes and returns a pointer to the allocated memory
  + *free(void \*p)*-frees the memory space pointed to by p, which must have been returned by a previous call to malloc() and other memory allocation functions
* Doug Lea’s malloc, or dlmalloc, memory manager implements the standard memory management functions
  + GNU C library and most versions of Linux are based on dlmalloc
  + Chunks of memory are maintained using a 'boundary tag' method where
    - Sizes of free chunks are stored both in the front of each chunk and at the end
    - Size fields hold P bits (short for PREV\_INUSE) representing whether previous chunks are free or in use
      * P=0, previous chunk is free
      * P=1, previous chunk is in use
    - Chunks always begin on even word boundaries
  + Tables 1-2 illustrate the general structure for allocated and free chunks
  + Memory pointer that is returned to users requesting allocated memory are labeled "mem" in Table 1
    - "mem" pointers are also on even word boundaries

Table -Allocated chunk

|  |
| --- |
| chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of previous chunk (if free) |  | or, user data from previous (last 8 bytes) |  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of chunk |P|  mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  . User data starts here... .  . .  . .  | |  nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | ...ends here (last 8 bytes) |  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of following chunk |1|  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |

Table -Free chunk

|  |
| --- |
| chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of previous chunk (if free) |  | or, user data from previous (last 4 bytes) |  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of chunk |P|  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Forward pointer to next chunk in bin |  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Back pointer to previous chunk in bin |  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  . Unused space (may be 0 bytes long) .  . .  | |  nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of previous chunk |  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  | Size of following chunk |0|  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |

* + Table 3 shows an example sequence of chunks
    - First, third, and fourth chunks are unallocated and in the same free-list bin
    - Second chunk is allocated

Table -Free list (bin) double-linked data structure

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Forward pointer to first chunk in bin |  | Size of previous chunk | | | | |
| Backward pointer to last chunk in bin |  | Size of current chunk | | | | 0 |
|  |  | Forward pointer to next | | | | |
|  |  | Backward pointer to previous | | | | |
| Head element |  | Unused space  ... | | | | |
|  |  | Size of previous chunk | | | | |
|  |  | Size of current chunk | | | 0 | |
|  |  | User data begins here... | | | | |
|  |  |  | | | | |
|  |  |  | | | | |
|  |  | ...ends here (last 8 bytes) | | | | |
|  |  | Size of current chunk | 1 | | | |
|  |  | Forward pointer to next | | | | |
|  |  | Backward pointer to previous | | | | |
|  |  | Unused space  ... | | | | |
|  |  | Size of previous chunk | | | | |
|  |  | Size of current chunk | | 0 | | |
|  |  | Forward pointer to next | | | | |
|  |  | Backward pointer to previous | | | | |
|  |  | Unused space  ... | | | | |

* + When memory is deallocated (i.e., free is called on it), the memory manager attempts to consolidate it with free chunks
    - When the chunk being deallocated has an unallocated chunk immediately before it
      * Previous free chunk is removed from the bin (unlinked) and consolidated with the chunk being deallocated
      * Resulting, consolidated chunk is placed in the bin appropriate to its size
    - Unlink occurs via the unlink macro in Table 4

Table -Unlink macro in malloc-2.7.2.c

|  |
| --- |
| /\* Take a chunk off a bin list \*/  #define unlink(P, BK, FD) { \  FD = P->fd; \  BK = P->bk; \  FD->bk = BK; \  BK->fd = FD; \  } |

* + Example unlink sequence is illustrated in Table 5
    - Free is called on the memory allocated in third chunk
    - Second chunk is unlinked as part of the consolidation in free
    - Step 1 assigns FD so that it points to the next chunk in the bin (wrt. chunk being unlinked)
    - Step 2 assigns BK so that it points to the previous chunk in the bin (wrt. chunk being unlinked)
    - Step 3 reassigns the back pointer of FD to the current chunk’s bk (i.e., one being unlinked)
      * I.e., replaces the backward pointer of the next chunk in the bin with the chunk pointed to by the current chunk’s bk
    - Step 4 reassigns the forward pointer of BK to the current chunk’s fd (i.e., one being unlinked)
      * I.e., replaces the forward pointer of the previous chunk in the bin with the chunk pointed to by the current chunk’s fd

Table -Chunk consolidation via unlink macro

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Size of previous chunk | |
|  |  | Size of current chunk | 0 |
|  | 4 | Forward pointer to next | |
|  |  | Backward pointer to previous | |
|  |  | Unused space  ... | |
|  | P | Size of previous chunk | |
| 1. FD=P->fd |  | Size of current chunk | 0 |
|  |  | Forward pointer to next | |
|  |  | Backward pointer to previous | |
| 2. BK=P->bk |  | Unused space  ... | |
| 3. FD->bk=BK |  | Size of previous chunk | |
| 4. BK->fd=FD |  | Size of current chunk | 0 |
|  |  | User data begins here... | |
|  |  |  | |
|  |  |  | |
|  |  | ...ends here (last 8 bytes) | |
|  |  | Size of current chunk | 1 |
|  |  | Forward pointer to next | |
|  | 3 | Backward pointer to previous | |
|  |  | Unused space  ... | |

* + Unlink macro in dlmalloc 2.7.2 is vulnerable to exploitation if the user can control the boundary tags
    - glibc-2.2.4 is based on this vulnerable code (see identical code in malloc/malloc.c around line 3180)

Table -Relevant portion of in malloc-2.7.2.c::free()

|  |
| --- |
| ...  nextchunk = chunk\_at\_offset(p, size);  nextsize = chunksize(nextchunk);  /\* consolidate backward \*/  if (!prev\_inuse(p)) {  prevsize = p->prev\_size;  size += prevsize;  p = chunk\_at\_offset(p, -((long) prevsize));  unlink(p, bck, fwd);  }  ... |

* Assume that an attacker could control the data within chunks that directly follow a chunk vulnerable to buffer overflow
  + Discuss in your group how one could take advantage of unlink by corrupting boundary tags

### Part II-Demo

* Download the project into a local sandbox

$ git clone <https://gitlab.com/underpantsgnomes/softwaresecurity/dlmallocexploit>

* Turn off address-space layout randomization in Ubuntu (ASLR) using

$ echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

* Ensure ASLR is disabled by confirming the output of running the following command is 0

$ cat /proc/sys/kernel/randomize\_va\_space

* Import the project in Eclipse
  + File->Open Projects From File System
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C compiler
  + Compile separately prior to a later linking step (-c)
  + Enable position independent code (-fPIC)
  + Specify to retain frame pointers (-fno-omit-frame-pointer); note: targeted leaf functions (ones that don’t call anything else) might not retain frame pointers even with this setting
  + Disable format string compile time warnings (-Wno-format-security); note: prevents build errors when –Wall is specified
  + Disable Ubuntu’s GCC-SSP enabling of run-time stack overflow verification via stack canaries (-fno-stack-protector)
  + Specify intel dialect for outputted assembly (-masm=intel)
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C linker
  + Enable executable stack for the linker (-z execstack)
* Inspect the exploit code in exploitWrapper.c
* Build the project
* Create the debug target configuration
* Select new launch configuration for a C/C++ Application
* Specify the wrapper binary built from the console as the target
* Insert the following three program arguments

|  |
| --- |
| ${string\_prompt: trampolineaddr:0x55555575c740} ${string\_prompt: oldrbp:0x7fffffffd860} |

* Go to the Debugger tab and specify an environment that uses ~/.gdbinit
* Set a breakpoint on line 44 in vulnerableUUT.c::vulnFunction()
* Run the demo
* Continue to the breakpoint and locate the stack frame for vulnerableUUT.c::vulnFunction()
  + Locate the local variables on the stack frame
  + Locate the previous rbp and return function pointer on the stack frame
* Two lines in malloc-2.7.2.c::free that performed chunk metadata validation were commented out to avoid difficulties with the exploit implementation
  + Line 3738: check\_inuse\_chunk(p);
  + Line 3808: check\_free\_chunk(p);
* Step through the vulnerability being exploited at the memcpy and free and discuss what is happening amongst your team.
* Locate the integer safety vulnerability
* How do the security features help to prevent exploitation?
* How would you correct the issues present in malloc-2.7.2.c?
  + E.g., see malloc-2.8.6.c version of unlink in Table 7

Table -Relevant portion of in malloc-2.7.2.c::free()

|  |
| --- |
| unlink\_small\_chunk(M, P, S) {  mchunkptr F = P->fd;  mchunkptr B = P->bk;  bindex\_t I = small\_index(S);  assert(P != B);  assert(P != F);  assert(chunksize(P) == small\_index2size(I));  if (RTCHECK(F == smallbin\_at(M,I) || (ok\_address(M, F) && F->bk == P))) {  if (B == F) {  clear\_smallmap(M, I);  }  else if (RTCHECK(B == smallbin\_at(M,I) ||  (ok\_address(M, B) && B->fd == P))) {  F->bk = B;  B->fd = F;  }  else {  CORRUPTION\_ERROR\_ACTION(M);  }  }  else {  CORRUPTION\_ERROR\_ACTION(M);  }  } |

## References

* <http://gee.cs.oswego.edu/dl/html/malloc.html>
* <http://www.openwall.com/advisories/OW-002-netscape-jpeg/>
* <http://gee.cs.oswego.edu/pub/misc/malloc-2.7.2.c>
* <http://gee.cs.oswego.edu/pub/misc/malloc-2.8.6.c>
* <https://ftp.gnu.org/gnu/libc/>
* Seacord, Robert C.. Secure Coding in C and C++, Pearson Education