Offensive Computer Security: Summary 3

David De Lille

April 21, 2015

1 Dynamic memory (continued)

1.1 Memory Allocator

The memory manager runs as part of the process. The linker adds in code provided by the OS, though compilers can provide alternatives. This code can be linked statically or dynamically.

The memory allocator needs 3 things to function: a list of available memory, an algorithm to allocate chunks (of a given size), and an algorithm to deallocate chunks. There are 2 methods for finding a suitable chunk for allocation: first fit (the first chunk that is big enough; faster) or best fit (the smallest chunk that is big enough; more efficient).

The free list can be thought of as a doubly-linked list of free chunks. Each chunk has boundary tags that contain metadata about the chunk, such as the size and the pointers to the previous/next chunks.

1.1.1 Doug Lea's dlmalloc allocator

This is the basis for Linux memory allocators. It arranges the free list as a circular, doubly-linked list of chunks (also called bins). Aside from the size of the chunk and the pointers to the surrounding chunks, the tag boundaries (for both used and freed chunks) contains either the size of the previous chunk (if free) or the last 4 bytes of the previous used chunk (if not free). Finally, it also contains a flag indicating if the previous flag is in use; this flag is stored as the last bit of the size of the chunk, because the size of chunks are always even (meaning the least significant bit would always be zero).

Because of the metadata that's stored for each chunk, the total used memory will always be smaller than the memory available to the program.

The unlinking algorithm (which removes a chunk from the free list for use by the program) works in 2 steps: it find the locations of the surrounding chunks, and then overwrite the pointers in those chunks to bypass the chunk in question.

See slides 11-18 for a visualisation. It's important to note that the system trusts the pointers provided.

The free algorithm is roughly the reverse of the unlinking algorithm, and it also involves overwriting pointers. During this algorithm, a check is performed to see if the next chunk happens to be free as well. If this is the case, the chunks are merged and returned to the free list as one. An example of freeing a chunk is illustrated in slides 19-26.

1.2 Heap vulnerabilities

1.2.1 Heap overflow

Slides 19-35 explain how an overflow on the heap can cause the free algorithm to overwrite an arbitrary memory location, provided by an attacker.

1.2.2 Double free

If free is called twice on the same chunk, the free list gets messed up and can lead to a vulnerability. There are some conditions, however:

- the chunk in question has to be isolated (both surrounding chunks have to be in use)
- the free list bin in which the chunk is going must be empty (all those size-chunks must be in use)

This bug is a lot more complicated than a heap overflow, and refers to the reference book (Secure Coding in C an C++, by Robert Seacord) for more information. This vulnerability affects dlmalloc and old version of RtlHeap, but most modern allocator alternatives do some sort of safe unlinking, which prevents most double frees.

1.2.3 Use after free

This happens when a pointer to a chunk is used after it has been freed. It is especially dangerous if it is used as a function pointer. This requires an attacker to overwrite the freed memory with a malicious substitute.

2 Integer security

2.1 Signed vs Unsigned

Unsigned values correspond to the 'normal' bit representation.

Signed numbers are stored in twos-complement. This means that positive numbers have the 'normal' bit representation, with a leading zero. Negative numbers have the reversed bit representation of the positive number that is 1 value

absolute value	positive	negative
0	00000000	N/A
1	00000001	11111111
2	00000010	11111110
...	...	...
126	01111110	10000010
127	01111111	10000001
128	N/A	10000000

Table 1: Bit representations of a signed char

smaller (e.g. -2 has the reversed bit representation of 1). The reason for this seemingly strange notation, is that addition because very simple and becomes similar to addition of unsigned numbers.

Note: Integers are signed by default

2.2 Integer truncation

Slides 42-43 show an example of an integer being truncated into a short. This is done (in this case) by placing the original value in eax, and then reading the ax register (which is smaller; see summary 2).

The standard data types can have different length depending on the system used. It is safer to instead use data types that explicitly call out the size and type of data stored in a variable.

2.3 Overflow

When an unsigned value is increased beyond the maximum, the result is always modulo, meaning it wraps around to smallest possible value (0).

Signed values can not overflow. This is undefined behaviour, according to the C99 standard, but some compilers still allow it.

2.3.1 Result saturation

Certain graphical processing units (GPUs) and digital processing units (DPUs) don't allow overflow. Instead, the result is going to be set to the max value. The reasoning is that if a value is representing, for example, how dark a pixel is, then increasing that value should not suddenly turn the highest possible value into the lowest possible value. It is better in that scenario to stop at the highest possible value. This, off course, does not conform to the C99 standard.

2.4 Underflow

This is the twin of overflow, that happens when a value is decreased below the minimum value. Again, the C99 standard dictates that only unsigned values can underflow; signed underflow is undefined behaviour.

2.5 Conversion/Promotion

When comparing a signed and an unsigned value, the signed value will converted to an unsigned value. This link contains more information on the rules for integer conversion.

For certain operations, the data will get promoted up to an integer, for the purpose of that operation.

2.6 Nuances

- -INT_MIN is considered undefined behaviour
- Bit shifting has certain limitations:
  - negative values can't be left shifted
  - a 1 can't be shifted into the sign position
  - a value can't be shift by more than the bitwidth of that object
- (int)x-1+1 can be undefined, if x = INT_MIN (but a compiler might optimise this out)
- a % b is undefined, if b is negative
- (short)x + 1 != (short)(x+1), if x = signed INT_ MAX

2.7 Casting

When casting a signed value to an unsigned value, negative numbers can become very big ((unsigned int)-1 == UINT_MAX).

There are 2 special data types reserved for storing sizes. size_t is equal to an unsigned int, because sizes should never be negative. However, some people wanted to be able to return -1 in certain functions to indicate an error, but because it would be casted to an unsigned int, this would lead to a very large size_t, which led to errors. The solution was another data type: ssize_t, which does allow for a -1 value (ssize_t $\in$ [-1, 32767]).

2.8 Important of integer bug

These types of bugs are often not understood by developers, and wrong use can lead to vulnerabilities. This paper explains overflow in C and C++ in more depth. Another issue that show the importance of these bugs, is that they can occur in cryptography libraries (as shown in this blog post).

2.9 Floats

Unlike integers, floats have a dedicated representation for special cases (Not A Number; NaN). Comparing this "value" to another float will always return zero, even if the other value is also NaN.

Precision is often lost during float calculations. Sums of floats don't always add up right. More nuances about precision in floats can be found here.

3 Format strings

Format strings are ways to control output for certain functions (printf, sprintf, snprintf, fprintf, syslog, ...). By using conversion specifiers, certain data can be represented in a specific way.

When an attacker can control a format string, this can lead to vulnerabilities; including Denial of Service, information leaks, and arbitrary writes. This technique can be used to defeat ASLR and enable Return Oriented Programming (ROP). Preventing these vulnerabilities comes down to not allowing user input into a format string.

3.1 Calling convention for functions

When a function is called, it's argument have to be passed. Some of the argument may get passed through registers, but when there are a lot of argument, they will get passed the stack. Slides 69-76 shows several examples for both 32-bit and 64-bit architectures.

In the end, the calling conventions depend on the architecture, the calling standard, and the type of function (system calls are called differently from normal code).

3.2 Conversion specifiers

%[flags][width][.precision][length-modifier] conversion-specifier

- %d or %i: signed decimal integer
- % u: unsigned decimal integer
- % o: unsigned octal

- %x: unsigned hexadecimal
- %X: unsigned hexadecimal (uppercase)
- %f: decimal float
- %e: scientific notation
- %a: hexadecimal floating point
- %c: char
- %s: string
- %p: pointer address
- %n: write number of characters written so far to corresponding address (nothing printed)

length	d, i	u,o,x,X	f,F,e,E, g,G,a,A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wwhar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

3.3 Exploiting format string vulnerabilities

This is just a short intro. Future summaries will go into more depth. Also, the book Hacking: The Art of Exploitation (HAOE) has a section (chapter 0x352) on format strings that explains the concept extremely well.

3.3.1 Denial of Service

format string = "%s%s%s%s%s%s%s%s...%s"
This format string will attempt to read a large number of strings off the stack, dereferencing values and interpreting them as pointers to strings. This will most likely cause a SEGFAULT, because of dereferencing non-pointer data (for example, NULL).

3.3.2 Information leaks

format string = "%08x %08x %08x %08x %08x %08x...%08x"
Printing this format string will print out the values on the stack (not dereferencing them), which could give an attacker insight into how the process works. It's important to remember that such values will be printed in little-endian.

3.3.3 View arbitrary memory locations

format string = "/xde/xf5/xe5/x04%x%x%x%x%s"
The previous format string can also be used to control the argument pointer, by moving it forward by 4 or 8 bytes. This can be exploited to view arbitrary memory locations.
Including the desired address (0x04e5f5de here) in the format string, will place it on the stack. After moving forward the argument pointer (the amount of %x's may vary), we can then read out that memory location (with %s).

3.3.4 Writing small values

format string = "/xde/xf5/xe5/x04%x%x%x%150x%n"
It is possible to use the %n conversion specifier to write small values to an arbitrary location, similar to how we can read values at an arbitrary location. By using a length modifier, this becomes even easier. However, it is still not easy to write larger values (such as memory addresses).

3.3.5 Arbitrary write

We can extend the previous case to be able to also write large values. Let's say we want to overwrite 4 bytes at a given address. Using a single write would not work, however, we could do it by writing 4 times; 1 byte at a time. It is easier to write 4 values smaller than 255 than 1 value, which can range up to 2^{32} (4294967296). More info on this in further summaries, or in chapter 0x352 of HAOE.

4 Concurrency and Race conditions

4.1 Concurrency and Parallelism

(Note: I don't agree with the definitions given by the course slides. Instead I use the definitions explained here.)

Concurrency is the composition of independently executing processes. Parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

There are 2 types of parallelism: data-centric and task-centric. Data-centric means the data is split up and processed in parallel (for example: a parallel implementation of MergeSort). Task-centric is when the work gets split up into distinct tasks that are executed in parallel.

4.2 Properties for Race conditions

- Concurrency: at least 2 independent control flows interacting
- Shared Object: same object shared by the concurrent flows
- Changing Object State: the object is changed by at least one of the concurrent flows

4.3 How to find Race conditions

- Focus on the shared objects
- Follow how each shared object is handled through the code. Focus on any state changes.
- When the state changes, what other flows could be interacting with it?

4.4 Consequences of Race conditions

Race conditions can result in corrupted values and inconsistent states. Volatile objects act in undefined ways when handled asynchronously. It is possible to abuse Race conditions to elevate permissions (e.g. CVE-2007-4302, CVE-2007-4303). Finally, a race condition can also lead to a deadlock (DoS).

5 Reading: HAOE 0x280 - 0x300 + 0x350 - 0x400

0x281 File access

File descriptor, file stream, file flags, bitwise operations

0x282 File permissions

Unix permissions (see summary 5), owner/group/other, chmod

0x283 User IDs

id, su, sudo, chsh, setuid, getuid/geteuid, passwd

0x284 Structs

Structs, time.h, epoch, member-of operator

0x285 Function pointers

Function pointers

0x286 Pseudo-random numbers

Pseudo-random, seeding with time()

0x351 Format parameters

Format parameters, %n, stack frame structure

0x352 Format string vulnerability

Format string vulnerability, examine stack memory, format string itself on stack

0x353 Reading from arbitrary memory addresses

Arbitrary read, read from environment variable

0x354 Writing to arbitrary memory addresses

Arbitrary write, field-width option, writing large values (one byte at a time), limitation of field-width option (it provides a minimum, but not a maximum)

0x355 Direct parameter access

Direct parameter access ("$")

0x356 Short writes

Length modifier ("h")

0x357 Detours with .dtors

.dtors table section, .ctors table section, destructor attribute, nm

0x358 Another notesearch vulnerability

Example of overwriting .dtors with address of env variable containing shellcode

0x359 Overwriting the Global Offset Table

Procedure Linkage Table (PLT), Global Offset Table (GOT)

6 Other notes

- Use the -m32 compiler flag to compile a 32-bit binary on a 64-bit system
- "/x" denotes a special, ASCII-encoded character (e.g. "/x61" corresponds to "a")
- http://q.viva64.com/: C++ bug finding quiz (60 seconds per bug)