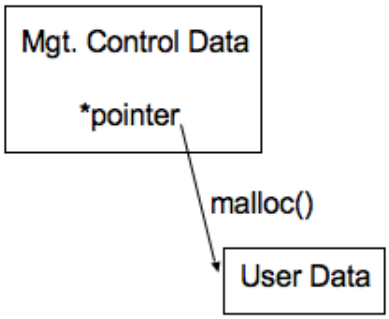
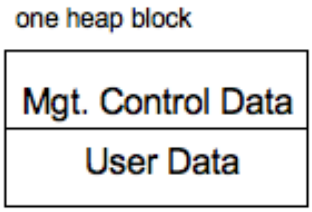
Heap Management

Before understanding how heap overflows work, is better to understand how heap management is done. One of the goals of heap management is to minimize the overhead of how information is stored. Heap management deals with a *free list,* of free blocks. One way of managing the free blocks is by keeping the Management Control Data separate from the User Data. When the user asks for memory, heap management allocates memory, gives it to the user, and keeps a pointer to this block of memory for bookkeeping purposes.



Another way is to have them both, Management Control Data and User Data, side by side in one data structure. Control Data preceding the User Data. In this way, one could imagine that heap management uses a double linked list to keep track of the free heap blocks.

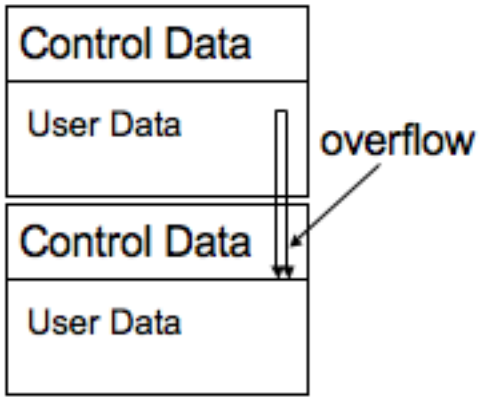


Keeping separate data structures complicates things. Allocation of data control structures is separate from allocation of user data structures; heap management has to keep track of more storage units. By putting the Control data and user data together in one block, one avoids this complexity, and reduces the level of indirection. This might help to the improvement in performance. For the rest of these notes, we will only consider the heap management implementation where control data and user data are kept in the same data structure. Most typical heap management systems implement this way of management too.

Heap Overflows

Let us say there is a heap block A, followed by another heap block B. When writing to a heap buffer in A, a heap overflow is possible if this buffer extends past the end of A and reaches into B. If it does, this buffer will overwrite control data in B.

There are many ways in which this type of overflow can be exploited, but they all are similar. An example of how this vulnerability can be exploited is by overwriting two fields in the control data section of an adjacent block. These two fields are next



and previous (remember free blocks are kept in a doubled linked list). The part of the heap management that adds and removes free blocks trusts the values referenced by next and previous, and this provides attackers with a way to mount an attack.

Suppose that an overflow occurs as shown in the above figure. Let us say that the overwritten control data belongs to a block in the freelist that is then allocated (as a result of a malloc that follows the copy operation that led to the overflow). To delete this block foo from the freelist, the heap management code might execute two statements:

foo->prev->next = foo->next;

foo->next->prev = foo->prev;

Focusing on the first statement, the effect of this statement under normal conditions (when the prev and next pointer fields stored in foo are valid) is shown in the figures below, where foo is the middle block in the chain.

Figure A

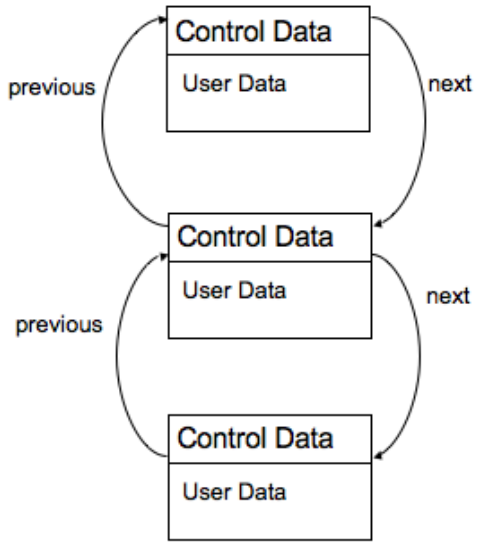
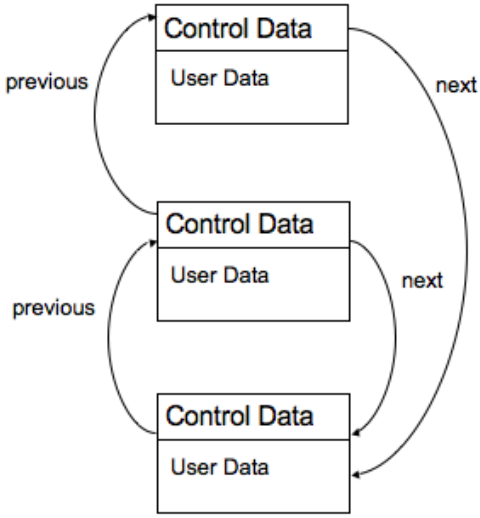


Figure B



However, if the attacker controls the values of these pointers (as a result of the overflow mentioned above) then the effect is quite different. In particular, assume that the attacker has overwritten the “prev” field with a value “a” and the next field with the value “b.” Then the effect of the italicized assignment above, when executed by the heap management code, is:

a->next = b,

Note that effect of the above statement is to store the value “b” at the address “a,” if “next” is the first field in the structure defining the heap blocks. (If it is not the first

field, then “0” will be replaced by a different constant value that represents the offset of the “next” field from the base of the structure.)

In effect, the heap management statement above allows the attacker to write an arbitrary word (“b”) at an arbitrary memory location (“a”). This can be used to carry out attacks. We point out that this corruption need not be triggered just by a malloc operation. It is possible to trigger similar attacks when a block is freed, or is merged with a preceding or succeeding blocks, although the exact details of the attack will differ.

Some of the possible targets that can be overwritten with a heap overflow attack:

1. **Function pointers** (code pointers) are the most attractive to attackers. A corrupted function pointer can point to code provided by the attacker and also already existing code.
    a. **Return Address** in a stack. In this case, the specific location of a return address is needed, and so, the canary mechanism used to detect a stack smashing attack will not work in this case. However, a second copy of the RA (return address) will continue to help.
    b. **Global Offset Table:** (dynamic linking) Table of function pointers in writable storage because it has to be filled in at run time. A two-step process is done to fill in this table. The program will not know where the library is, the library location will be known at run time, when the library is loaded. A table is constructed with n slots, where n is the number of functions used by the program. Say the program calls function f(), and the third slot in the table is assigned to f(). So the call of f() will be replace with call *func_table[3]. At run time, the dynamic linker fills in the table. Attackers want to execute system calls, so an attacker can overwrite a GOT entry corresponding to commonly called functions such as “read” so that it points to the attacker's code.
    c. **Function Pointers in static memory.**

2. **Data Pointers**
    a. Name of programs executed or files opened.
    b. Application specific data, for example: a login program uses different methods of authenticating a user within a loop, and when one of these methods succeeds, the program may set a variable named “is_authenticated.” When this variable becomes true, the program may break out of this loop and proceed to execute a shell. An attack can change the value of this flag so the program breaks out of the loop and logs in the attacker.

Defenses against heap overflow attacks

Heap canaries: put a canary between two heap blocks. Depending upon the implementation details, the canary at the end of each heap block, at the beginning of blocks, or in the middle of the control data. This is what most typical systems do: Linux, Windows. This kind of protection is not very expensive.

In general, separating user data from control data is a good idea, it makes the program less vulnerable and harder to exploit, and avoid the idea of an attacker changing the flow of control of a program, which can be more powerful than an attack that just changes the data. We mentioned the example of separating heap control data from heap user data, but the same high level concept can be thought as being operational in the context of stack-canaries, or ProPolice.

Between data attacks and injected code attacks, attackers might choose injected code, but attacks on data can also be very powerful, as described in the previous example of changing the login program.

Format String Attacks

printf, printing to a file buffer
sprintf, write to a character array

These functions take variable number of arguments. The callee does not know how many parameters were passed. The callee has to figure out how many arguments were passed by looking at the format argument, and read them from the stack. If an attacker controls the format argument, she can fool the printf routine to read an arbitrary number of arguments from the stack. Moreover, since the format string controls the operation of the printf-family of functions, this control basically allows the attacker to exert significant control over the code executed by the victim program.

A format string vulnerability exists in programs that read a string from an untrusted source (e.g., a socket connection with a remote site) into a variable *s* and uses a statement

printf(*s*)

to print it, instead of the more secure form

printf("%s", s)

Now, by providing the value of "s, the attacker can control the effect of printf.
, or by making it think that a different number of parameters than the actual number of parameters are being passed. Still, not all problems are solved. In particular, the attacker wants to write something into memory (e.g., overwrite return address) in order to gain control, but printf only reads from memory. Well, almost. There is an obscure "%n" format directive that involves writing to a specified memory location. But the data written is not directly controllable: it is the number of characters successfully printed so far. But the attacker can control this value by setting the field widths while printing. Moreover, by using a "hh" prefix to "n," the attacker can write just the least significant byte of the number of characters printed. For example, the attacker can write "50" into location x and a "30" into location x+1 using the following format string, provided that he can arrange the %n parameters to reference the values "x" and "x+1" that must be on the stack.

"<u>%50d %hhn %206</u> %30d %hhn"

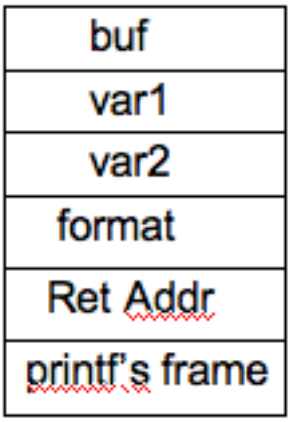
256

How does the attacker ensure that "x" and "x+1" appear on the stack. Usually,

format string vulnerabilities are exploitable when the format string argument *s* described above resides on the stack. For example, consider the following vulnerable function:

```
void f() {
   char buf[256]
   int var1
   int var2
   read(d, buf, 256)
   printf(buf)
  ...
```

printf interprets whatever is on the stack as parameters. When printf is called we look at the stack. The "format" variable is the first parameter taken by the printf function. When f calls prints, it would have stored the location of *buf* in that location. Assuming that the compiler allocates variables in the order in which they were declared, we might see the local variables of the caller (i.e., function f) above this parameter.
The way parameter passing works, printf will interpret the words in the stack as parameters. Thus, the location corresponding to var2 will be interpreted as the second argument, var1 as 3rd argument and so on.



Note that the attacker can start referencing *buf* as the 4th parameter – from this point on, the attacker controls the addresses corresponding to the %n argument.

Format string vulnerabilities are very specific and can be easily avoided. Just look for printf calls where there is just one argument to printf, and make a quick check whether the data being read is a static string, or is it being read from somewhere else. A more general defense would ensure that the variable argument feature is being used securely, e.g., the calller can specifically send in another argument that indicates the number of parameters.

Integer Overflow

There're multiple forms of integer overflows:
Assignments between variables of different width. E.g. assign a 32-bit value to a 16-bit variable. In this case, the higher 16-bit will be discarded.

Assign an unsigned integer to a signed integer variable. If the unsigned value has "1" on the highest bit, after the assignment, the signed variable will be an negative integer.
An integer overflow can cause heap overflow if you allocate less memory space than needed.