This is a very, *very* brief guide to a few common types of memory corruption attacks. Almost all details have been ommitted, but it should give you the basic ideas.

Buffer Overruns

Most memory corruption attacks start with some type of **buffer overrun**. A buffer overrun is when a program allocates some amount memory to a buffer—say 100 bytes—and then writes beyond the end of that allocated memory.

Buffer overruns can be done on the stack...

```
void f1a(void * arg, size_t len) {
  char buff[100];
  memcpy(buff, arg, len); /* buffer overrun if len > 100 */
  /* ... */
  return;
}
```

...or on the heap.

```
void f1b(void * arg, size_t len) {
  char * ptr = malloc(100);
  if (ptr == NULL) return;
  memcpy(ptr, arg, len); /* buffer overrun if len > 100 */
  /* ... */
  return;
}
```

(Both of these examples were taken from the paper "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns" by Pincus and Baker.)

The examples above are written in C. C provides no built-in mechanisms to prevent writing past the end of a buffer (unlike languages such as Java, where you'd expect to see some sort of ArrayIndexOutOfBounds exception).

Stack Smashing

One of the original techniques to exploit buffer overruns was **stack smashing**. To understand stack smashing, you first have to understand how a stack works.

When a piece of code calls a function f, the compiler pushes a few things onto the stack before it makes the jump to f. In particular, it pushes the current address onto the stack. When the function f is ready to return, the compiler uses that address as the return address, so that the code returns to the calling function.
To smash the stack, an attacker cleverly overwrites a buffer so that it overwrites the return address. The new return address points to some code that the attacker has written. This means that, when f returns, it will jump to the malicious code rather than the calling function.

(The original stack smashing "paper" is not required reading for 6.033, but it is pretty enjoyable.)

Return-to-libc

Basic stack-smashing attacks—where the return address is overwritten to point to code that the attacker wrote—can be thwarted by making the stack non-executable (the attacker's code will exist on the stack, so if the stack is marked as non-executable, that code can't run). This technique is sometimes known as W⊕X.

However, in that case, attackers can overwrite return addresses to point to some existing subroutine—often a shared library—that is executable. Those shared libraries typically contain subroutines for making system calls. If an attacker overwrites the return address, as well as some other carefully selected locations on the stack, they can pass parameters to these system calls and execute arbitrary code. These types of attacks are known as return-to-libc attacks (libc being a very common shared library).

Return-oriented Programming

Return-oriented programming (ROP) builds on the idea from return-to-libc attacks. Instead of returning to a particular subroutine in a shared library, ROP allows attackers to return to some small chunk of code from shared libraries (rather than a full subroutine). These chunks of code—"gadgets"—can be chained together to allow for arbitrary code execution.