Memory safety — Attacks and Defenses

In the first few lectures we will be looking at software security—problems associated with the software implementation. You may have a perfect design, a perfect specification, perfect algorithms, but still have implementation vulnerabilities. In fact, after configuration errors, implementation errors are probably the largest single class of security errors exploited in practice.

We will start by looking at a particularly prevalent class of software flaws, those that concern *memory safety*. Memory safety refers to ensuring the integrity of a program's data structures: preventing attackers from reading or writing to memory locations other than those intended by the programmer.

Because many security-critical applications have been written in C, and because C has peculiar pitfalls of its own, many of these examples will be C-specific. Implementation flaws can in fact occur at all levels: in improper use of the programming language, the libraries, the operating system, or in the application logic. We will look at some of these others later in the course.

1 Buffer overflow vulnerabilities

We'll start with one of the most common types of error—*buffer overflow* (also called *buffer overrun*) vulnerabilities. Buffer overflow vulnerabilities are a particular risk in C. Since it is an especially widely used systems programming language, you might not be surprised to hear that buffer overflows are one of the most pervasive kind of implementation flaws around.

As a low-level language, we can think of C as a portable assembly language. The programmer is exposed to the bare machine (which is one reason that C is such a popular systems language). A particular weakness that we will discuss is the absence of automatic bounds-checking for array or pointer access. A buffer overflow bug is one where the programmer fails to perform adequate bounds checks, triggering an out-of-bounds memory access that writes beyond the bounds of some memory region. Attackers can use these out-of-bounds memory accesses to corrupt the program's intended behavior.

Let us start with a simple example.

```
char buf[80];
void vulnerable() {
    gets(buf);
}
```

In this example, `gets()` reads as many bytes of input as are available on standard input, and stores them into `buf[]`. If the input contains more than 80 bytes of data, then `gets()` will write past the end of `buf`, overwriting some other part of memory. This is a bug. This bug typically causes a crash and a core-dump. What might be less obvious is that the consequences can be far worse than that.

To illustrate some of the dangers, we modify the example slightly.

```
char buf[80];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

Imagine that elsewhere in the code, there is a login routine that sets the `authenticated` flag only if the user proves knowledge of the password. Unfortunately, the `authenticated` flag is stored in memory right after `buf`. If the attacker can write 81 bytes of data to `buf` (with the 81st byte set to a non-zero value), then this will set the `authenticated` flag to true, and the attacker will gain access. The program above allows that to happen, because the `gets` function does no bounds-checking: it will write as much data to `buf` as is supplied to it. In other words, the code above is *vulnerable*: an attacker who can control the input to the program can bypass the password checks.

Now consider another variation:

```
char buf[80];
int (*fnptr)();
void vulnerable() {
    gets(buf);
}
```

The function pointer `fnptr` is invoked elsewhere in the program (not shown). This enables a more serious attack: the attacker can overwrite `fnptr` with any address of their choosing, redirecting program execution to some other memory location. A crafty attacker could supply an input that consists of malicious machine instructions, followed by a few bytes that overwrite `fnptr` with some address A. When `fnptr` is next invoked, the flow of control is redirected to address A. Notice that in this attack, the attacker can choose the address A however they like—so, for instance, they can choose to overwrite `fnptr` with an address where the malicious machine instructions will be stored (e.g., the address `&buf[0]`). This is a *malicious code injection* attack. Of course, many variations on this attack are possible: for instance, the attacker could arrange to store the malicious code anywhere else (e.g., in some other input buffer), rather than in `buf`, and redirect execution to that other location.

Malicious code injection attacks allow an attacker to seize control of the program. At the conclusion of the attack, the program is still running, but now it is executing code chosen by the attacker, rather than the original software. For instance, consider a web server that receives requests from clients across the network and processes them. If the web server contains a buffer overrun in the code that processes such requests, a malicious client would be able to

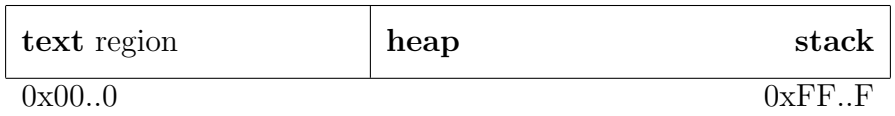
seize control of the web server process. If the web server is running as root, once the attacker seizes control, the attacker can do anything that root can do; for instance, the attacker can leave a backdoor that allows them to log in as root later. At that point, the system has been "*owned*".

Buffer overflow vulnerabilities and malicious code injection are a favorite method used by worm writers and attackers. A pure example such as we showed above is relatively rare. However, it does occur: for instance, it formed the vectors used by the first major Internet worm (the *Morris Worm*). Morris took advantage of a buffer overflow in `in.fingerd` (the network "finger" daemon) to overwrite the filename of a command executed by `in.fingerd`, similar to the example above involving an overwrite of an "authenticated" flag. But pure attacks, as illustrated above, are only possible when the code satisfies certain special conditions: the buffer that can be overflowed must be followed in memory by some security-critical data (e.g., a function pointer, or a flag that has a critical influence on the subsequent flow of execution of the program). Because these conditions occur only rarely in practice, attackers have developed more effective methods of malicious code injection.

2 Stack smashing

One powerful method for exploiting buffer overrun vulnerabilities takes advantage of the way local variables are laid out on the stack.

We need to review some background material first. Let's recall C's memory layout:



The **text** region contains the executable code of the program. The **heap** stores dynamically allocated data (and grows and shrinks as objects are allocated and freed). Local variables and other information associated with each function call are stored on the **stack** (which grows and shrinks with function calls and returns). In the picture above, the **text** region starts at smaller-numbered memory addresses (e.g., 0x00..0), and the **stack** region ends at larger-numbered memory addresses (0xFF..F).

Function calls push new stack frames onto the stack. A stack frame includes space for all the local variables used by that function, and other book-keeping information used by the compiler for this function invocation. On Intel (x86) machines, the stack grows down. This means that the stack grows towards smaller memory addresses. There is a special register, called the stack pointer (SP)[1], that points to the beginning of the current stack frame. Thus, the stack extends from the address given in the SP until the end of memory, and pushing a new frame on the stack involves subtracting the length of that frame from SP.

[1] In our discussion here, we use a simpler description, and more common terminology, for how this works. In section and lecture, we will bridge from this to the actual terminology and mechanisms used on x86 systems, because the class project will require you to have familiarity with those specifics.

Intel (x86) machines have another special register, called the instruction pointer (IP), that points to the next machine instruction to execute. For most machine instructions, the machine reads the instruction pointed to by IP, executes that instruction, and then increments the IP. Function calls cause the IP to be updated differently: the current value of IP is pushed onto the stack (this will be called the return address), and the program then jumps to the beginning of the function to be called. The compiler inserts a function prologue—some automatically generated code that performs the above operations—into each function, so it is the first thing to be executed when the function is called. The prologue pushes the current value of SP onto the stack and allocates stack space for local variables by decrementing the SP by some appropriate amount.

When the function returns, the old SP and return address are retrieved from the stack, and the stack frame is popped from the stack (by restoring the old SP value). Execution continues from the return address.

After all that background, we're now ready to see how a *stack smashing* attack works. Suppose the code looks like this:

```
void vulnerable() {
    char buf[80];
    gets(buf);
}
```

When `vulnerable()` is called, a stack frame is pushed onto the stack. The stack will look something like this:

If the input is too long, the code will write past the end of `buf` and the saved SP and return address will be overwritten. This is a *stack smashing* attack.

Stack smashing can be used for malicious code injection. First, the attacker arranges to infiltrate a malicious code sequence somewhere in the program's address space, at a known address (perhaps using techniques previously mentioned). Next, the attacker provides a carefully-chosen 88-byte input, where the last four bytes hold the address of the malicious code.[2] The `gets()` call will overwrite the return address on the stack with the last 4 bytes of the input—in other words, with the address of the malicious code. When `vulnerable()` returns, the CPU will retrieve the return address stored on the stack and transfer control to that address, handing control over to the attacker's malicious code.

The discussion above has barely scratched the surface of techniques for exploiting buffer overrun bugs. Stack smashing was first introduced in 1996 (see "Smashing the Stack for Fun and Profit" by Aleph One). Modern methods are considerably more sophisticated and powerful. These attacks may seem esoteric, but attackers have become highly skilled at

[2] In this example, I am assuming a 32-bit architecture, so that SP and the return address are 4 bytes long. On a 64-bit architecture, SP and the return address would be 8 bytes long, and the attacker would need to provide a 96-byte input whose last 8 bytes were chosen carefully to contain the address of the malicious code.

exploiting them. Indeed, you can find tutorials on the web explaining how to deal with complications such as:

- The malicious code is stored at an unknown location.
- The buffer is stored on the heap instead of on the stack.
- The attack can only overflow the buffer by a single byte.[3]
- The characters that can be written to the buffer are limited (e.g., to only lowercase letters).[4]
- There is no way to introduce *any* malicious code into the program's address space.

Buffer overrun attacks may appear mysterious or complex or hard to exploit, but in reality, they are none of the above. Attackers exploit these bugs all the time. For example, the *Code Red* worm compromised 369K machines by exploiting a buffer overflow bug in the IIS web server. In the past, many security researchers have underestimated the opportunities for obscure and sophisticated attacks, only to later discover that the ability of attackers to find clever ways to exploit these bugs exceeded their imaginations. Attacks once thought to be esoteric to worry about are now considered easy and routinely mounted by attackers. The bottom line is this: *If your program has a buffer overflow bug, you should assume that the bug is exploitable and an attacker can take control of your program.*

How do you avoid buffer overflows in your code? One way is to check that there is sufficient space for what you will write before performing the write.

3 Format string vulnerabilities

Let's look next at another type of vulnerability:

```
void vulnerable() {
    char buf[80];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

Do you see the bug? The last line should be `printf("%s", buf)`. If `buf` contains any `%` characters, `printf()` will look for non-existent arguments, and may crash or core-dump the program trying to chase missing pointers. But things can get much worse than that.

[3] In one of the most impressive instances of this, the Apache webserver at one point had an off-by-one bug that allowed an attacker to zero out the next byte immediately after the end of the buffer. The Apache developers downplayed the impact of this bug and assumed it was harmless—until someone came up with a clever and sneaky way to mount a malicious code injection attack, using only the ability to write a single zero byte one past the end of the buffer.

[4] Imagine writing a malicious sequence of instructions, where every byte in the machine code has to be in the range 0x61 to 0x7A ('a' to 'z'). Yes, it's been done.

If the attacker can see what is printed, the attacker can mount several attacks:

- The attacker can learn the contents of the function's stack frame. (Supplying the string `"%x:%x"` reveals the first two words of stack memory.)
- The attacker can also learn the contents of any other part of memory, as well. (Supplying the string `"%s"` treats the next word of stack memory as an address, and prints the string found at that address. Supplying the string `"%x:%s"` treats the next word of stack memory as an address, the word after that as an address, and prints what is found at that string. To read the contents of memory starting at a particular address, the attacker can find a nearby place on the stack where that address is stored, and then supply just enough `%x`'s to walk to this place followed by a `%s`. Many clever tricks are possible, and the details are not terribly important for our purposes.) Thus, an attacker can exploit a format string vulnerability to learn passwords, cryptographic keys, or other secrets stored in the victim's address space.
- The attacker can write any value to any address in the victim's memory. (Use `%n` and many tricks; the details are beyond the scope of this writeup.) You might want to ponder how this could be used for malicious code injection.

The bottom line: *If your program has a format string bug, assume that the attacker can learn all secrets stored in memory, and assume that the attacker can take control of your program.*

4 Integer conversion vulnerabilities

What's wrong with this code?

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 80) {
        error("length too large: bad dog, no cookie for you!");
        return;
    }
    memcpy(buf, p, len);
}
```

Here's a hint. The prototype for `memcpy()` is:

```
void *memcpy(void *dest, const void *src, size_t n);
```

And the definition of `size_t` is:

```
typedef unsigned int size_t;
```

Do you see the bug now? If the attacker provides a negative value for `len`, the `if` statement won't notice anything wrong, and `memcpy()` will be executed with a negative third argument. C will cast this negative value to an `unsigned int` and it will become a very large positive integer. Thus `memcpy()` will copy a huge amount of memory into `buf`, overflowing the buffer.[5]

Note that the C compiler won't warn about the type mismatch between `signed int` and `unsigned int`; it silently inserts an implicit cast. This kind of bug can be hard to spot. The above example is particularly nasty, because on the surface it appears that the programmer has applied the correct bounds checks, but they are flawed.

[5] In an earlier draft of these notes, the conditional was written as "`if (len > sizeof buf)`". A student astutely observed that that code would actually be okay, because `sizeof buf` is of type `unsigned`, and the rules of C would first convert `len` to also be `unsigned`. So a negative value of `len` would instead be treated as a very large `unsigned` value, and the test would succeed, leading to printing of the error message and a return before the call to `memcpy`. In the above verision, we instead use the constant `80`, which C treats as `signed`, so a negative value of `len` will sneak past the test.

Here is another example. What's wrong with this code?

```
void vulnerable() {
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len+5);
    read(fd, buf, len);
    ...
}
```

This code seems to avoid buffer overflow problems (indeed, it allocates 5 more bytes than necessary). But, there is a subtle problem: `len+5` can wrap around if `len` is too large. For instance, if `len = 0xFFFFFFFF`, then the value of `len+5` is `4` (on 32-bit platforms). In this case, the code allocates a 4-byte buffer and then writes a lot more than 4 bytes into it: a classic buffer overflow. You have to know the semantics of your programming language very well to avoid all the pitfalls.

5 Memory safety

Buffer overflow, format string, and the other examples above are examples of *memory safety* bugs: cases where an attacker can read or write beyond the valid range of memory regions. Other examples of memory safety violations include using a dangling pointer (a pointer into a memory region that has been freed and is no longer valid) and double-free bugs (where a dynamically allocated object is explicitly freed multiple times). C and C++ rely upon the programmer to preserve memory safety, but bugs in the code can lead to violations of memory safety. History has taught us that memory safety violations often enable malicious code injection and other kinds of attacks.

Some modern languages are designed to be intrinsically memory-safe, no matter what the programmer does. Java is one example. Thus, memory-safe languages eliminate the opportunity for one kind of programming mistake that has been known to cause serious security problems.

6 Defending against memory-safety vulnerabilities

We've only scratched the surface of implementation vulnerabilities. If this makes you a bit more cautious when you write code, then good! We'll finish with a discussion of some techniques to defend against memory-safety vulnerabilities. Here are five general classes of defensive techniques::

- **Secure coding practices.** We can adopt a disciplined style of programming that avoids these bugs.

- **Better languages/libraries.** We can adopt languages or libraries that make such mistakes harder to commit.
- **Runtime checking.** We can have our compiler (or other tools) automatically inject runtime checks everywhere they might be needed to detect and prevent exploitation of memory-safety bugs, so that if our code does have a memory-safety bug, it won't be exploitable by attackers.
- **Static analysis.** We can use a compiler (or a special tool) to scan the source code and identify potential memory-safety bugs in the code, and then task developers with fixing those bugs.
- **Testing.** We can applying testing techniques to try to detect memory-safety bugs in the code, so that we can fix them before the software ships to our customers.

Many of these techniques can be applied to all kinds of security bugs, but for concreteness, let's explore how they can be applied to protect against memory-safety vulnerabilities.

6.1 Secure Coding Practices

In general, before performing any potentially unsafe operation, we can write some code to check (at runtime) whether the operation is safe to perform and abort if not. For instance, instead of

```
char digit_to_char(int i) { // BAD
    char convert[] = "0123456789";
    return convert[i];
}
```

we can write

```
char digit_to_char(int i) { // BETTER
    char convert[] = "0123456789";
    if (i < 0 || i > 9)
        return "?"; // or, call exit()
    return convert[i];
}
```

This code ensures that the array access will be within bounds. Similarly, when calling library functions, we can use a library function that incorporates these kinds of checks, rather than one that does not. Instead of

```
char buf[512];
strcpy(buf, src); // BAD
```

we can write

```
char buf[512];
strlcpy(buf, src, sizeof buf); // BETTER
```

The latter is better, because `strlcpy(d,s,n)` takes care to avoid writing more than `n` bytes into the buffer `d`. As another example, instead of

```
char buf[512];
sprintf(buf, src); // BAD
```

we can write

```
char buf[512];
snprintf(buf, sizeof buf, src); // BETTER
```

Instead of using `gets()`, we can use `fgets()`. And so on.

In general, we can check (or otherwise ensure) that array indices are in-bounds before using them, that pointers are non-null and in-bounds before dereferencing them, that integer addition and multiplication won't overflow or wrap around before performing the operation, that integer subtraction won't underflow before performing it, that objects haven't already been de-allocated before freeing them, and that memory is initialized before being used.

We can also follow *defensive programming* practices. Defensive programming is like defensive driving: the idea is to avoid depending on anyone else around you, so that if anyone else does something unexpected, you won't crash. Defensive programming is about surviving unexpected behavior by other code, rather than by other drivers, but otherwise the principle is similar.

Defensive programming means that each module takes responsibility for checking the validity of all inputs sent to it. Even if you "know" that your callers will never send you a NULL pointer, you check for NULL anyway, just in case, because sometimes what you "know" isn't actually true, and even if it is true today, it might not be true tomorrow as the code evolves. Defensive programming is about minimizing trust in the other components your code interacts with, so that the program fails gracefully if one of your assumptions turns out to be incorrect. It's usually better to throw an exception or even stop the program than to allow a malicious code injection attack to succeed. Don't write fragile code; strive for robustness. Defensive programming might lead to duplicating some checks or introducing some unnecessary checks, but this may be a price worth paying for reducing the likelihood of catastrophic security vulnerabilities.

It can also be helpful to perform code reviews, where one programmer reviews the code written by another programmer to ensure (among other things) that the code follows secure code practices. For instance, some companies have a requirement that all code be reviewed by another programmer before being checked in. To help code reviewers (and yourself), a good principle is to organize your code so that it is obviously correct. If the correctness of your code would not be obvious to a reviewer, rewrite it until its correctness is self-evident. As the great programmer Brian Kernighan once wrote:

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

In the case of avoiding memory-safety bugs, code should be rewritten until it is self-evident that it never performs any out-of-bounds memory access, for instance by introducing explicit runtime checks before any questionable memory access.

One potential issue with relying solely upon secure coding practices is that we are still relying upon programmers to never make a mistake. Programmers are human, and so errors are inevitable. The hope is that secure coding practices can reduce the frequency of such errors and can make them easier to spot in a code review or debugging session. Also, static analysis tools (see below) may be able to help detect deviations from secure coding practices, further reducing the incidence of such errors.

Many secure coding practices have a strong overlap with good software engineering principles, but the demands of security arguably place a heavier burden on programmers. In security applications, we must eliminate *all* security-relevant bugs, no matter how unlikely they are to be triggered in normal execution, because we face an intelligent adversary who will gladly interact with our code in abnormal ways if there is profit in doing so. Software reliability normally focuses primarily on those bugs that are most likely to happen; bugs that only come up under obscure conditions might be ignored if reliability is the goal, but they cannot be ignored when security is the goal. Dealing with malice is harder than dealing with mischance.

6.2 Better Languages and Libraries

Languages and libraries can help avoid memory-safety vulnerabilities by eliminating the opportunity for programmer mistakes. For instance, Java performs automatic bounds-checking on every array access, so programmer error cannot lead to an array bounds violation. Also, Java provides a String class with methods for many common string operations. Importantly, every method on String is memory-safe: the method itself performs all necessary runtime checks and resizes all buffers as needed to ensure there is enough space for strings. Similarly, C++ provides a safe string class; using these libraries, instead of C's standard library, reduces the likelihood of buffer overflow bugs in string manipulation code.

6.3 Runtime Checks

Compilers and other tools can reduce the burden on programmers by automatically introducing runtime checks at every potentially unsafe operation, so that programmers do not have to do so explicitly. For instance, there has been a great deal of research on augmenting C compilers so they automatically emit a bounds check at every array or pointer access.

One challenge is that automatic bounds-checking for C/C++ has a non-trivial performance overhead. Even after decades of research, the best techniques still slow down computationally-intensive code by 10%-150% or so, though the good news is that for I/O-bound code the overheads can be smaller.[6] Because many security-critical network servers are I/O-bound,

[6] If you're interested in learning more, you can read a research paper on this subject at http://www.usenix.org/events/sec09/tech/full_papers/akritidis.pdf.

automatic bounds-checking may be feasible for some C/C++ programs. The reason that bounds-checking is expensive for C is that one must bounds-check every pointer access, which adds several instructions for the check for each instruction that accesses a pointer, and C programs use pointers frequently. Also, to enable such checks, pointers have to carry information about their bounds, which adds overhead for the necessary book-keeping.

Another potential issue with automatic bounds-checking compilers for C/C++ has to do with legacy code. First, the source code of the program has to be available, so that the program can be re-compiled using a bounds-checking compiler. Second, it can be difficult to mix code compiled with bounds-checking enabled with libraries or legacy code not compiled in that way. To check the bounds on a pointer `p`, we need some way to know the start and end of the memory region that `p` points into, which requires either changing the memory representation of pointers (so that they are an address, a base, and an upper bound) or introducing extra data structures. Either way, this poses challenges when code compiled in this way interacts with code compiled by an older compiler.

There are other techniques that attempt to make it harder to exploit any memory-safety bugs that may existing the code. You might learn about some of them in discussion section.[7]

6.4 Static Analysis

Static analysis is a technique for scanning source code to try to automatically detect potential bugs. You can think of static analysis as runtime checks, performed at compile time: the static analysis tool attempts to predict whether there exists any program execution under which a runtime check would fail, and if it finds any, it warns the programmer. Sophisticated techniques are needed, and those techniques are beyond the scope of this class, but they build on ideas from the compilers and programming language literature for automatic program analysis (e.g., ideas initially developed for compiler optimization).

The advantage of static analysis is that it can detect bugs proactively, at development time, so that they can be fixed before the code has been shipped. Bugs in deployed code are expensive, not only because customers don't like it when they get hacked due to a bug in your code, but also because fixes require extensive testing to ensure that the fix doesn't make things worse. Generally speaking, the earlier a bug is found, the cheaper it can be to fix, which makes static analysis tools attractive.

One challenge with static analysis tools is that they make errors. This is fundamental: detecting security bugs can be shown to be undecidable (like the Halting Problem), so it follows that any static analysis tool will either miss some bugs (false negatives), or falsely warn about code that is correct (false positives), or both. In practice, the effectiveness of a static analysis tool is determined by its false negative rate and false positive rate; these two can often be traded off against each other. At one extreme are verification tools, which are guaranteed to be free of false negatives: if your code does not trigger any warnings, then it is guaranteed to be free of bugs (at least, of the sort of bugs that the tool attempts to detect).

[7] For those interested in reading more, you can read about ASLR and the NX bit.

In practice, most developers accept a significant rate of false negatives in exchange for finding some relevant bugs, without too many false positives.

6.5 Testing

Another way to find security bugs proactively is by testing your code. A challenge with testing for security, as opposed for functionality, is that security is a negative property: we need to prove that nothing bad happens, even in unusual circumstances; whereas standard testing focuses on ensuring that something good does happen, under normal circumstances. It is a lot easier to define test cases that reflect normal, expected inputs and check that the desired behavior does occur, then to define test cases that represent the kinds of unusual inputs an attacker might provide or to detect things that are not supposed to happen.

Generally, testing for security has two aspect:

1. *Test generation.* We need to find a way to generate test cases, so that we can run the program on those test cases.
2. *Bug detection.* We need a way to detect whether a particular test case revealed a bug in the program.

Fuzz testing is one simple form of security testing. Fuzz testing involves testing the program with random inputs and seeing if the program exhibits any sign of failure. Generally, the bug detection strategy is to check whether the program crashes (or throws an unexpected exception). For greater bug detection power, we can enable runtime checks (e.g., automatic array bounds-checking) and see whether any of the test cases triggers a failure of some runtime check. There are three different approaches to test generation that are commonly taken, during fuzz testing:

- *Random inputs.* Construct a random input file, and run the program on that input. The file is constructed by choosing a totally random sequence of bytes, with no structure.
- *Mutated inputs.* Start with a valid input file, randomly modify a few bits in the file, and run the program on the mutated input.
- *Structure-driven input generation.* Taking into account the intended format of the input, devise a program to independently "fuzz" each field of the input file. For instance, if we know that one part of the input is a string, generate random strings (of random lengths, with random characters, some of them with `%` signs to try to trigger format string bugs, some with funny Unicode characters, etc.). If another part of the input is a length, try random integers, try a very small number, try a very large number, try a negative number (or an integer whose binary representation has its high bit set).

One shortcoming of purely random inputs is that, if the input has a structured format, then it is likely that a random input file will not have the proper format and thus will be quickly rejected by the program, leaving much of the code uncovered and untested. The other two approaches address this problem. Generally speaking, mutating a corpus of valid files is

easier than writing a test generation suite customized to the particular input format under consideration, but may be less effective.

Usually, fuzz testing involves generating many inputs: e.g., hundreds of thousands or millions. What it sacrifices in sophistication, it makes up for in quantity. Fuzz testing is popular in industry today because it is cheap, easy to apply, and somewhat effective at finding some kinds of bugs (more effective than you would think it has any right to be).[8]

[8] For those interested in learning more, check out *Real World Fuzzing*, http://pages.cs.wisc.edu/~rist/642-fall-2012/toorcon.pdf. If you like to try things out, you could experiment with zzuf, an easy-to-use mutation fuzzer for Linux: http://caca.zoy.org/wiki/zzuf. For instance, on Linux you can have some fun with a command like `zzuf -v -s 1:1000 valgrind -q --leak-check=no --error-exitcode=1 unzip -o foo.zip`; look for error messages from Valgrind, prefixed with `==NNNNN==`.