cmsc330

Cybersecurity

Cybersecurity Breaches

Major security breaches of computer systems are a fact of life. They affect companies, governments, and individuals. Focusing on breaches of individuals' information, consider just a few examples:

- Equifax (2017) - 145 million consumers’ records
- Adobe (2013) - 150 million records, 38 million users
- eBay (2014) - 145 million records
- Anthem (2014) - Records of 80 million customers
- Target (2013) - 110 million records
- Heartland (2008) - 160 million records

Vulnerabilities: Security-relevant Defects

The causes of security breaches are varied but many of them, including those given above, owe to a **defect** (or *bug*) or **design flaw** in a targeted computer system's software. The software problem can be **exploited** by an attacker. An **exploit** is a particular, cleverly crafted input, or a series of (usually unintuitive) interactions with the system, which trigger the bug or flaw in a way that helps the attacker.

Kinds of Vulnerability

One obvious sort of vulnerability is a bug in security policy enforcement code. For example, suppose you are implementing an operating system and you have code to enforce access control policies on files. This is the code that makes sure that if Alice's policy says that only she is allowed to edit certain files, then user Bob will not be allowed to modify them. Perhaps your enforcement code failed to consider a corner case, and as a result Bob is able to write those files even though Alice's policy says he shouldn't. This is a vulnerability.

A more surprising sort of vulnerability is a bug in code that seems to have nothing to do with enforcing security all. Nevertheless, by exploiting it the attacker can break security anyway. Exploits of such bugs have fun names such as **buffer overflow**, **SQL injection**, **cross-site scripting (XSS)**, **command injection**, among many more. We'll talk about these and others in this part of the course.

Correctness vs. Security

The bugs underlying security exploits are, under normal circumstances, irritating. A normal user who stumbles across them will experience a crash or incorrect behavior.

But a key thing to observe is that **an attacker is not a normal user!**

A *normal user* will attempt to *avoid* triggering software defects, and when they do trigger them, the *effects* are oftentimes *benign*.

An *attacker* will actively attempt to *find* defects, using unusual interactions and features. When they find them, they will try to *exploit* them, to *deleterious effect.*

Code that is material to a system's secure operation is called its **trusted computing base** (TCB). This is the code that *must* be correct if the security of the entire system is to be assured. Not all code is part of the TCB. For example, perhaps your implementation of the `ls` command in your operating system mis-formats some of its output. This is irritating, but it doesn't compromise the operating system's security.

Thus, while it's expected that software will have bugs even after it's deployed (perfection is expensive!), bugs in the TCB are more problematic. These are bugs that an attacker will exploit to compromise your system, causing potentially significant damage. And these bugs can be in surprising places.

Building Security In

What can we do about software that has vulnerabilities?

Adding Security On

Security companies like Symantec, McAfee, FireEye, Cisco, Kaspersky, Tenable, and others propose to deal with vulnerabilities by providing a service that runs alongside target software. Their approach is to *add a layer of security on top of the software, separate from it.*

For example, you can buy services that monitor your system and prevent vulnerabilities in resident software from being exploited. Or these services may attempt to mitigate the impact of exploitation. For example, a very specific kind of input may be required to exploit a vulnerability, and the security monitoring system can look for inputs like it, as defined by a *signature*. If an input matches the signature, the security system can block or modify it and thus stop the exploit.

But there are two problems with this approach:

- It is *retrospective*. It can only work once the vulnerability, and an exploit for it, is known. Then we can make a signature that blocks the exploit. Until the vulnerability is discovered and a signature is made for it, attackers will be able to successfully (and surreptitiously) exploit it.
- It is *not (always) general*. Oftentimes there are many possible inputs that can exploit a vulnerability, and they may not all be described with an efficiently recognizable signature. As such, an attacker can bypass the monitoring system by making a change to the exploit that does not match the signature but still achieves the desired effect.

In sum: While security monitoring is useful, it has not proven it to be effective at (completely) solving the security problem. There is more to do.

Designing and Building for Security, from the Start

A more direct (and complementary) way of dealing with vulnerabilities is to avoid introducing them in the first place, when designing and writing the software. We call this approach **Building Security In**. It requires that every software developer (this means you!) knows something about security and writes their code with a security mindset.

At a high level, building a system with a security mindset involves doing two things:

- **Model threats**. We need to think a little bit about how an attacker could influence or observe how our code is run. The process will clarify what we can assume about the trustworthiness of inputs our code receives, and the visibility of outputs our code produces (or how it produces them).

- **Employ Defensive *Design Patterns***. Based on the identified threats, we apply tried-and-true *design patterns* to defend against them. A design pattern is a kind of code template that we customize to the specifics of our software or situation. Sometimes this code pattern can be automatically introduced by a language, compiler, or framework, or we may need to write the code and customize the template ourselves.

In the rest of this lecture, we will cover three broad topics

- The basics of *threat modeling*.
- Two kinds of *exploits*: **Buffer overflows** and **command injection**. Both of these are instances of the more general exploit pattern of **code injection**, which we see again in the next lecture in the form of *SQL injection* and *cross-site scripting*.
- Two kinds of *defense*: **Type-safe programming languages**, and **input validation**. The use of the former blocks buffer overflows and many other exploits. The latter is a design pattern often programmed by hand which aims to ensure that data influenced by a potential attacker cannot force our code to do the wrong thing. It defends against buffer overflows (when not already using a type-safe language), command injection, and many other code injection exploits.

Threat Modeling

Avoiding attacks means understanding what attackers can do, and putting mechanisms in place to stop them from doing things you don't want. Broadly speaking, this is the process of *threat modeling*.

The Lovely and Dangerous Internet, in Brief

For those interested in widespread, diverse, and innovative services and information, the Internet is an incredible blessing. For those aiming to defend services from attack, the Internet is a curse. This is because the Internet makes services available to just about anyone, oftentimes without any sort of accountability – potential attackers can probe and prod many Internet services essentially without risk. Understanding what they can do is important for writing software accessible via the Internet. If you think your software won't be connected to the Internet, you might think again; sooner or later it will end up there.

Clients and Servers

Most sessions over the internet can view as a simplified interaction between clients and servers.

- The *client* seeks to use a network-connected services, e.g., using a web browser, ssh client, specialized app, etc. to exchange messages with that service, over the Internet
- Services are provided by a remote machine called the *server.* Oftentimes there is a pool of such machines, but the client won't necessarily see that — they just visit store.com, not realizing that each interaction might go to a different server on a different machine.

Both the client and server have resources that need protection.

The client may be on a machine with personal financial information, private records, browsing history, etc. The server will host valuable information as part of the service it is offering, such as bank account balances, student grades, personnel records, etc. Even the computing resources themselves may be valuable; e.g., an attacker could attempt to surreptitiously use a client's machine to send spam or carry out illicit activities.

Security Policies

The client and server resources warrant, roughly speaking, three kinds of protection

- **Confidentiality**. A confidentiality (aka privacy) policy states that a resource should not be readable by just anyone. Only certain users/actors should have access to the information. For example, my personal information, whether stored at a client or server, should be kept private, and not be leaked.
- **Integrity**. An integrity policy indicates which parties may add to or modify a resource. For example, my bank account balance should be changeable only by me or the bank, and the change should only be via agreed-upon mechanisms, e.g., deposits or withdrawals, or certain fees.
- **Availability**. An availability policy states that protected resources should not be unreasonably inaccessible. Having money at the bank but then not being able to access it for a long period would be a violation of availability. If an attacker managed to get an account on my machine and started using it to send spam, that would also be a violation of availability, because now (some of) my machine is not available for my own use.

We will see that by exploiting software vulnerabilities, attackers are able to violate any and all of these policies.

Threats to (Internet-connected) Computer Systems

What are the ways that an attacker can cause problems? These three are typical.

As a Normal Client

An attacker can interact with an Internet service via the same channel allotted to normal users. If a normal user of an on-line store browse the store's products, make purchases, leave reviews, submit support requests, etc. then so can an attacker.

Of course, an attacker will attempt to perform these actions in weird and wonderful ways in an attempt to exploit potential vulnerabilities. Attackers will not limit themselves to using standard tools and will not follow instructions. For example:

- Instead of using an Internet browser, an attacker may craft network messages by hand, perhaps ones that are too long or use unusual characters.
- Instead of always following the expected process, the attacker will try things that are off the beaten path. For example, rather than go to store.com via a browser, and then type in a username/password, and then click a link to go to a page, the attacker may skip the login page entirely and just type in the URL of the desired page, to attempt to bypass the authentication process.

Thus, designers and implementers of computer systems need to think about the limits of what is possible, not just what is expected, to properly defend against a malicious client.

As a "Man in the Middle" (MITM)

Normal clients can send messages to and from the server, but are normally oblivious to the interactions the server might be having with other clients. A more powerful kind of attacker can observe such interactions and potentially corrupt or manipulate them. Traditionally, such an attacker is called a Man in the Middle (MITM) because they can sit in between two (or more) communicating parties.

This sort of situation is more common than you might think because of the prevalence of wireless networks. It is not hard to observe messages being sent around on that network, since it uses a broadcast medium. If you are connected to the public network at a coffee shop, then someone else connected to that same network may be looking at the messages you send and receive.

The typical defense against this kind of attacker is to use cryptography. By using *encryption*, even if the attacker can see your messages, they can't make sense of them (ensures confidentiality). By using *digital signatures* (*aka* message authentication codes, or *MAC*s), even if the attacker can modify your messages, the receiver will be able to detect the modifications (ensures integrity). Additional defenses are also sometimes required, e.g., the use of *nonces* to detect the potential *replay* of messages.

We will not discuss cryptography or MITM-style attacks in this course. If you are interested in learning more, check out CMSC 456 and CMSC 414.

As a Co-resident Service/user

Sometimes an attacker can do more than just communicate via the same network as the target; they can have direct access to the same machine.

In multi-user operating systems, such as Linux or MacOs, an attacker can log in as one user in an attempt to steal or manipulate information belonging to another. In today's cloud computing environments, like Microsoft's Azure or Amazon's Web Services (AWS), a target may be running a virtual machine instance on the same host as one run by the attacker. Mobile code technologies enable attacker-provided code to run directly on the target machine. For example, an attacker could try to upload a malicious Javascript program to a web site that is visited by a potential victim, who will unknowingly download and run the program in their browser when they visit the site.

In these cases, co-residency means an attacker can observe or modify resources shared or even directly owned by the target, such as memory or stable storage (files), in such a way as to achieve his goals.

Buffer Overflows

What is a Buffer Overflow?

A *buffer overflow* describes a family of possible exploits of a vulnerability in which a program may incorrectly access a buffer outside its allotted bounds. A *buffer overwrite* occurs when the out-of-bounds access is a write. A *buffer overread* occurs when the access is a read.

The key question is: **What happens when an out-of-bounds access occurs?**

If your program is written in a memory-safe or type-safe programming language, some important attacks are taken off the table; if it is not, then there are fairly dire security implications, as we will see.

Example: Out-of-Bounds Read/write in OCaml

Consider the following simple program, written in OCaml. There's a bug in this code; can you spot it?

```
let rec incr_arr x i len =
  if i >= 0 && i < len then
    (x.(i) <- x.(i) + 1;
    incr_arr x (i+1) len)
;;

let y = Array.make 10 1;;
incr_arr y 0 (1 + Array.length y);;
```

The function `incr_arr` iterates over an OCaml array, adding `1` to each of its elements. The code creates an array `y` of size 10, with each element initialized to 1. It then calls `incr_arr` on this array. What happens?

```
Exception: Invalid_argument "index out of bounds".
```

The bug is on line 8: the code should be pass `Array.length y` as the third argument to `incr_arr`, but is instead passes that expression, plus 1. As such, `incr_arr` will read (and attempt to write) one past the end of the array, which the program detects and signals by throwing an exception.

Example: Out-of-Bounds Read/write in C

Here's our buggy example again but this time written in C.

```c
 1  #include <stdio.h>
 2
 3  void incr_arr(int *x, int len, int i) {
 4    if (i >= 0 && i < len) {
 5      x[i] = x[i] + 1;
 6      incr_arr(x,len,i+1);
 7    }
 8  }
 9
10  int y[10] = {1,1,1,1,1,1,1,1,1,1};
11  int z = 20;
12
13  int main(int argc, char **argv) {
14    incr_arr(y,11,0);
15    printf("%d =? 20\n",z);
16    return 0;
17  }
```

As in the OCaml version, the code in `main` invokes `incr_arr` with array `y`, but after calling the function, `main` also prints out the value of another variable, `z`. As in the OCaml version, the array `y` that is being passed to `incr_arr` has length 10, but the second argument is given as 11, which is one more than it should be. As a result, line 5 will be allowed to read/write past the end of the array.

In our OCaml version, running the program resulted in an exception being thrown; what will happen in this C version?

I can compile this program with the `clang` C compiler (no optimizations) on my Mac to produce the executable `a.out`. When I run `a.out`, it completes normally, outputting the following:

```
21 =? 20
```

What happened?

Basically, when `incr_arr` read `x[10]` on line 5 (i.e., when argument `i == 10`), the function read the contents (`20`) of `z`, since it happens to be allocated in memory just past the end of `y` (the memory that `x` is pointing to). Then line 5 added 1 to what it read, and wrote the result (`21`) back to `z`, which is printed by `main`.

When we fix the bug (by changing line 14 to be `incr_arr(y,10,0);`), the program prints the following instead, since `z` is no longer modified:

```
20 =? 20
```

To recap: In OCaml, reading/writing outside the bounds of a buffer *always* results in an exception, and the program halts. In C, reading/writing outside the bounds of the buffer is *not* guaranteed to cause an exception. Instead, basically anything could happen. Here, we saw that adjacent memory can be read or modified.

Exploiting a Vulnerability

When an application has a bug in it that allows the program to access a buffer outside its bounds, an attacker can try to exploit it. He does this by controlling the input to the program in a way that triggers the bug to his advantage.

In our example, the program always fails, so that's not very interesting. But what if we changed `main` to be the following ?

```
1  #include <stdlib.h>
2  int main(int argc, char **argv) {
3    int len = 10;
4    if (argc == 2) len = atoi(argv[1]);
5    incr_arr(y,len,0);
6    printf("%d =? 20\n",z);
7    return 0;
8  }
```

If we run the compiled program `a.out` with no arguments, then it works as expected (the guard on line 4 is false). But suppose we run it thusly.

```
a.out 11
```

Since `argc == 2`, the guard on line 4 is true so it sets `len` to be 11. Thus the call to `incr_arr` on line 5 is the same buggy one we saw in the original version of our buggy C program, and `z` will be overwritten.

An attacker can exploit the bug if he can influence what (and whether) the argument is passed to `a.out`. If he can force the argument to be 11 (or more), then he can trigger the bug.

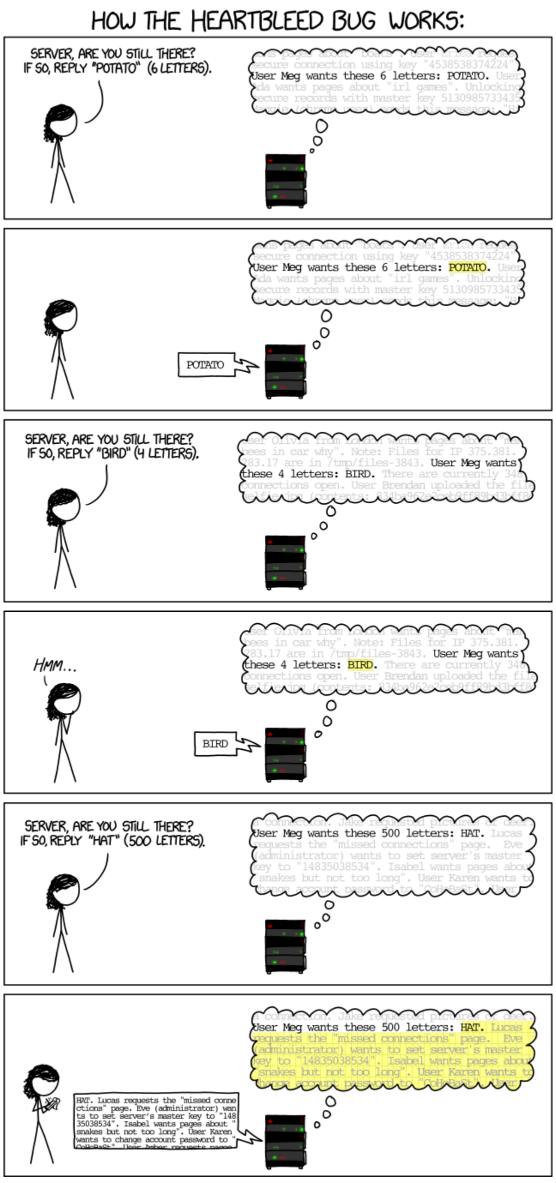
What Can Exploitation Achieve?

To recap: Suppose my program has a bug in it that allows reading or writing outside the bounds of a buffer, and the attacker is able to control that happening, at least to some degree. What can the attacker achieve?

Buffer Overread: Heartbleed

Heartbleed is a bug in the popular, open-source OpenSSL codebase that is used by a substantial number of Internet clients and servers to carry out secure communication, e.g., as part of the HTTPS protocol. By one account, Heartbleed "made more headlines and news articles in one day than any war had since Vietnam." Now it has its own website!

This XKCD comic explains it very well:


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
secure connection using key "4538538374224". User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file
HMM...
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

Heartbleed's operation is not so different than our previously developed example.

In that example, the `a.out` program can be invoked with an argument that specifies the claimed length. As we saw, the length could be wrong! This means that if the attacker is able to run `a.out` with a given length that's greater than the actual length of the buffer (i.e., 11 or more), then the attacker can cause the buffer to be read (and written) outside its bounds, accessing nearby memory when that happens.

In Heartbleed, the attacker (the person, in the XKCD cartoon) acts as the client. It sends a "heartbeat" message to the victim server (the machine, in the cartoon). This message includes some text, and specifies the length of the text. The machine writes the client's text to a buffer and then sends a message back with the contents of the buffer, which contains that text. Crucially, the machine asks the client to specify a length along with the text, and this length can be wrong! Just as with our example, if the specified length is greater than the actual length of the provided text, the server will just send back whatever happens to be in memory beyond the end of the buffer.

As it turns out, this memory could contain very sensitive information, such as secret cryptographic keys or passwords, perhaps provided by previous clients. Due to OpenSSL's popularity, many Internet servers needed to be shut down and patched, and public cryptographic key certificates that they used to authenticate their identity needed to be reissued, out of fears that exploits of Heartbleed may have stolen them.

Buffer Overwrite: Morris Worm

The Morris Worm was the work of Robert Morris, a computer science student at Cornell University. Occurring in 1988, it is one of the first widely acknowledged (and impactful) uses of a buffer overflow exploit (in the program fingerd) for the purposes of injecting code.

As described in this article, the exploit worked by sending a larger message to the `fingerd` program than it expected to ever receive. In particular, the program pre-allocated a buffer of size 500 in a local variable in `main`. The exploit sent to the program an input that filled that buffer with 536 bytes of data – more than it could hold. As a result, the extra bytes spilled out and overwrote other parts of the call stack, including `main`'s return address. Therefore, when the `main` function completed, instead of properly returning to its caller, it "returned" to an address placed by the attacker.

Cleverly, the exploit chose the overwrite the return address with a value that pointed *inside the just-overflowed buffer*, so that after the return, the contents of that buffer were treated by the machine as code. This code was also cleverly chosen so that when executed it performs `execve("/bin/sh",0,0)` ; i.e., it turns the current program into a shell running on the attacked system. This attack came to be known as a *stack smashing attack* and the attack's payload came to be known as shellcode.

Buffer overwrites are also dangerous even when stack smashing (or similar sorts of code injection attacks) are not possible. Returning to our example, imagine that the variable `z` represented a flag that indicates whether a user has been authorized to carry out a sensitive operation. By overflowing buffer `y` , an unauthorized user will be granted excess privilege.

Code Injection

The Morris Worm exploit aims to perform **code injection**. It works by tricking the program to treat attacker-provided data as code. Buffer overflows are one vector to doing code injection, where the injected code is machine code. But it is not the only one, as we will see later; others include

- SQL injection
- Command injection
- Cross-site scripting
- Use-after-free (violating Temporal Safety)
- ... and many others are a kind of bug, coupled with an exploit that aims to inject code of the attacker's choice.

How can we defeat them?

Defense: Type-safe Languages

To recap what we have seen so far about buffer overflows:

- A buffer overflow is the term for a family of exploits of bugs that allow a buffer to be read or written outside its bounds.
- A buffer overflow can be used to steal sensitive information (as with Heartbleed) or even to inject and run attacker-provided code (as with the Morris Worm).

We also saw something very interesting: *Neither of these, nor indeed any other, exploits of buffer overflows are possible in OCaml*. Why? Because any attempt to read or write outside the bounds of a buffer is detected immediately, and met with a run-time exception.

This is a consequence of OCaml being a ***type-safe* programming language**; indeed, the programs written in Java and Ruby are immune from buffer overflows and similar sorts of memory corruption attack, too, since they are also type-safe.

Why Is Type Safety Helpful?

Type safety ensures two useful properties that preclude buffer overflows and other memory corruption-based exploits.

Type safety ensures that memory in use by the program at a particular type *T always* has (or can be treated as having) that type *T*. This property, called **preservation** (aka *subject reduction*), ensures that, for example, if `x` is a pointer to an integer (`int ref` in OCaml), it will *always* be usable at that type. No action by the program can change `x` 's contents to break the invariants assumed of values of that type. As a consequence, this means that a pointer-typed value cannot be overwritten to make it something that is not a proper pointer (created by a legitimate program action) to the program's heap or stack.

In addition, values deemed to have type *T* will be usable by code expecting to receive a value of that type; i.e., the aforementioned invariants of values of a type T are sufficient to ensure safe execution. For example, code expecting to receive an array of integers will not break if it tries to iterate down the array and increment each element, since these operations (reading and writing within the stated bounds of the array, and adding one to an integer) are permitted on objects of this type. This property is called **progress**.

To ensure preservation and progress implies that buffers can only be accessed within their allotted bounds, precluding buffer overflows. This is because overwrites could break preservation, since they could end up breaking the assumed invariants of the type of a nearby value. Overreads could break progress, since an out-of-bounds read may return memory that does not conform to the invariants of the expected type (e.g., reading an out of the bounds of an array of integer pointers may return gibberish and not a proper pointer).

Type safety turns out to preclude other pernicious exploits, too, that work by corrupting memory. These include **use after free** (an example of which we give below), **format string attacks**, and **type confusion**, among others. These exploits attempt to force memory created at type *T* to be used at another type *S* instead. By enforcing type safety, the language ensures these attacks can't happen. This is good for program security, and it's also good for program reliability and even programmer productivity, since type safety rules out many sorts of hard-to-find bugs that make a program prone to crashing.

Costs of Ensuring Type Safety

C and C++ are the only programming languages in common use that eschew type safety. Why do they do this? There are two commonly cited reasons, **performance** and **expressiveness**. There are two direct forms of performance slowdown typical of type-safe languages: **bounds checks** and **garbage collection**.

Array Bounds Checks for Spatial Safety

Run-time checks to ensure that buffer accesses are in bounds (thus enforcing *spatial safety*) add overhead to a program's running time. If these checks occur in a tight loop, their impact is multiplied. Reconsider our OCaml example:

```
let rec incr_arr x i len =
  if i >= 0 && i < len then
    (x.(i) <- x.(i) + 1;
    incr_arr x (i+1) len)
```

In principle, safely executing `x.(i)` in `incr_arr` requires the compiler to add a lower-bound check and an upper-bound check, to confirm that `i >= 0` and `i <` *length(* `x` *).* This pair of checks will be inserted for each `x.(i)` occurring on line 3, i.e., for both the read and the write. Since not much else is happening in the loop, the bounds checks could dominate running time.

Fortunately, it is now common for a compiler to avoid inserting bounds checks when it can prove they are not needed. This is true of ahead-of-time compilers for languages like OCaml, and the just-in-time compilers that occur inside of language virtual machines, like the Java Virtual Machine (such as the one maintained by Oracle) or the Javascript execution engine (e.g., like Google's V8).

In our example, the OCaml compiler knows that to reach line 3, the guard on line 2 must have been true and thus that `i >= 0 && i < len`. Knowing this, we can deduce that as long as `len <=` *length(* `x` *)*, the access `x.(i)` will be in bounds, so this is the check the compiler will insert. Moreover, since neither `i` nor `x` are changed between the two `x.(i)` accesses on line 3, there is no need to perform the check twice: One check will do, just prior to line 3, for both accesses. Such reasoning allows the compiler to replace the four checks it would have inserted with a single one. Here's what the program would look like, if we made the check explicit:

```
let rec incr_arr_with_check x i len =
  if i >= 0 && i < len then
    (if len <= Array.length x then
       x.(i) <- x.(i) + 1
     else raise (Invalid_argument "index out of bounds");
    incr_arr_with_check x (i+1) len)
```

Even this check can be eliminated with a little more work: The compiler just has to add a check that `len <=` *length(* `x` *)* prior to the initial call to `incr_arr`; it will be trivially true for each of the recursive calls. (And, likewise, if we'd written our program as above with an explicit check, the compiler will know that no additional checks will be needed, since we've done the required checking already.)

Recent work on an extension to C called **Checked C** aims to support bounds checking in C, to avoid buffer overruns. Preliminary work (in 2018 and 2020) has shown that running time overhead due to bounds checks can be reduced to just a few percent, on average, if the compiler is smart enough.

Temporal Safety Violations

In C and C++ it is the programmer's responsibility to free dynamically allocated memory that is no longer needed, so that it can be recycled. In C, dynamic memory is allocated via a call to `malloc`, and this memory is freed by passing its pointer to `free`. In C++ one calls `new` to allocate an object and `delete` to free it. Both calls are typically inexpensive.

In a type-safe language, manual allocation may be explicit (like `new` in Java and Ruby) while programmer-directed deallocation, e.g., via `free` or `delete`, is generally disallowed (with Rust being a notable exception). This is because prematurely deallocating memory can compromise safety, in particular *temporal safety*. To see why, consider the following C program.

```c
#include <stdlib.h>
struct list {
  int v;
  struct list *next;
};
int main() {
  struct list *p = malloc(sizeof(struct list));
  p->v = 0;
  p->next = 0;
  free(p); // deallocates p
  int *x = malloc(sizeof(int)*2); // reuses p's old memory
  x[0] = 5; // overwrites p->v
  x[1] = 5; // overwrites p->next
  p = p->next; // p is now bogus
  p->v = 2; // CRASH!
  return 0;
}
```

The top of the program defines a `struct list` for a linked list; the value stored at the list node is in field `v`, and the pointer to the next list element is in the `next` field. The `main` function calls `malloc` to allocate a `struct list` node, storing a pointer to it in `p`, and then lines 8 and 9 initialize the value and next pointer to `0` (recall that in C, `0` when used as a pointer is equivalent to `null` in Java). Line 10 frees this memory, allowing it to be reused.

Then the trouble begins. Line 11 calls `malloc` which is very likely to reuse the just-freed memory. Here, `malloc` creates a 2-element `int` array, stored in `x`, and lines 12 and 13 initialize each of the array's elements to `5`. Line 14 is buggy: It uses `p` even though `p` no longer points to valid memory, since it was freed; `p` is called a **dangling pointer**. Assuming that `x` and `p` point to the same memory, due to the memory being recycled, `x[0]` is now an *alias* for `p->v` and `x[1]` is an alias of `p->next`. This means that writing to `x[1]` on line 13 overwrites `p->next` with `5`. The overwrite on line 13 means that when line 14 updates `p` to `p->next` it is setting `p` to `5`, and when line 15 performs `p->v`, it is dereferencing `5` as if it were a pointer. Since `5` is very unlikely to be a legitimate memory address (usually the lower range of memory is reserved for the operating system), and is certainly not a pointer to a proper `struct list` as its type claims, we have violated type safety. On my Mac, if we compile and run this program we see:

```
Segmentation fault: 11
```

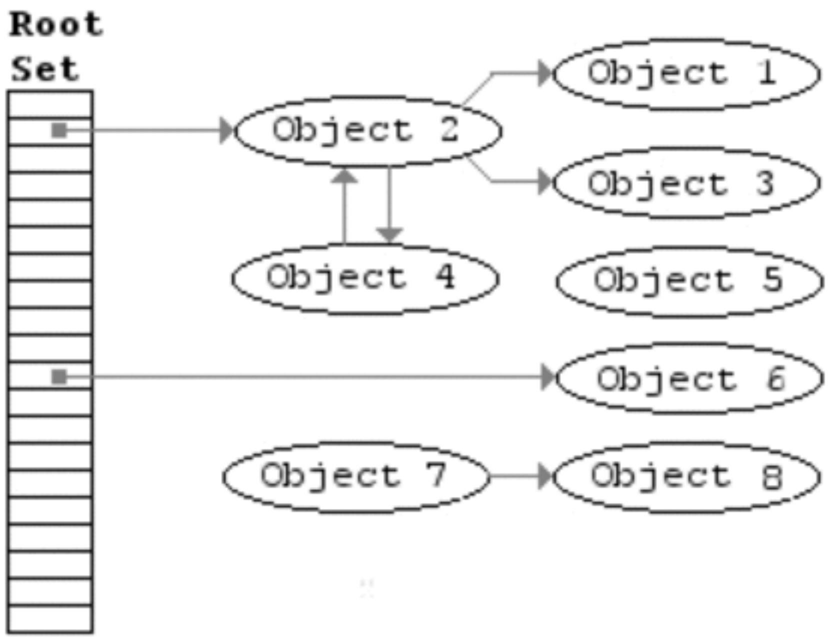
Using a pointer after it is freed is a bug that can be (and frequently is) exploited, e.g., to perform code injection. Such an exploit is, aptly, referred to as a **use after free**. Such exploits are not possible in a type-safe language, which the language usually ensures by precluding calls to `free` entirely. If you can't call `free` to deallocate the memory, you can't deallocate it prematurely! But if we are not allowed to call `free` how do we avoid running out of usable memory?

Garbage Collection

Garbage collection (GC) is, generally speaking, an automatic process by which dynamically allocated memory that the program no longer needs is identified and recycled. A GC works by reclaiming any objects that are not *reachable* by the actively running program. A reachable object is one that can be accessed via a chain of pointers, starting from a local or global variable. A reachable object *may* be accessed by the program in the future, so it's not safe to get rid of it. On the other hand, an unreachable object will *never* be accessed, since there's no path to it from any in-scope variables; hence, it's safe to remove.

Reachability is typically determined by one of two methods: **tracing** or **reference counting**.

A tracing GC determines reachability directly: Starting from the local and global variables (the "roots"), the GC follows any pointers it comes across, ultimately identifying all of the reachable memory objects. Unreached objects are recycled.



Tracing in a Mark/Sweep Garbage Collector

[Above is from https://en.wikipedia.org/wiki/Tracing_garbage_collection#/media/File:Animation_of_the_Naive_Mark_and_Sweep_Garbage_Collector_Algorithm.gif but it is not embedding properly.]

Reference counting determines reachability incrementally, as the program runs: Along with each object is stored a hidden *reference count* field, which keeps track of the number of direct pointers to the object that exist in the program. Every time a pointer is updated to point to an object, that object's reference count is incremented. When a pointer to an object is dropped, e.g., by the program changing the pointer or it going out of scope and/or being freed, the count Is decremented. When a count goes to zero, the object is garbage and can be recycled.

Garbage collection helps ensure type safety by avoiding the use-after-free bugs we saw before: No object will be freed if it could be used again in the future. This is good for security, and it’s also good for ease of use—figuring out when to free an object is one less thing the programmer needs to worry about.

On the other hand, GC adds extra performance overheads. A tracing collector adds to the program’s running time the cost of periodically tracing the program’s memory and collecting the unreachable objects. Tracing is also problematic for performance because the program is paused while it takes place. Tracing collectors can reduce running time overhead by delaying when tracing takes place, but doing so adds space overhead—the longer you wait to trace, the more garbage there is taking up potentially valuable memory. In more advanced GCs, tracing can also be done in parallel, to speed up GC while the program is paused, and/or concurrently with the program while it runs, thus slowing its progress but not stopping it during GC.

A reference counting collector avoids this long pause but still adds overhead for maintaining reference counts (and can result in a periodic pause if there are cascading deletes). Reference counting also adds a per-object space overhead, due to having to store the count.

The time and space overheads of garbage collection are often perfectly acceptable, given the productivity and security benefits GC offers. This fact is evidenced by the vast number of programs now written in garbage collected languages; back in the early to mid 90s, GC was a dirty word in many circles! But there are some circumstances for which performance is at a premium, and in these circumstances there is resistance to moving away from C and C++, despite the risks.

Expressiveness

Type safety, as we have seen, is often enforced by limiting the operations allowed on particular objects to ones that (always) ensure progress and preservation (and, usually, a form of type abstraction, which we won't get into). This means that type safety can sacrifice needed expressiveness in some cases.

For example, the C programming language was originally invented at AT&T Bell Labs to enable writing the UNIX operating system in a high-level language (i.e., not machine code). In operating system code, you are interacting with the underlying hardware. You may know that a particular hardware address must store a pointer to a structure, say the page tables; the operating system needs to be able to store this pointer at that address. So, the OS code casts that address to a pointer to a `struct` that describes the format of the page tables.

This sort of operation – cast from integer to pointer – is not permitted in a type safe language. We have already seen why: Not every integer is a legal pointer, and the unfettered cast permits many dangerous operations. But in the OS context, we need to allow this sort of thing at least sometimes. So, C was designed to afford programmers more freedom to perform potentially dangerous actions.

C similarly permits casting between different sorts of objects, e.g., a `struct` and an array. We do this during calls to `memcpy`, for example, which takes a pointer to a buffer:

```c
#include <string.h>

struct foo { int x; int y; };

void f(struct foo *p) {
  struct foo s;
  memcpy(&s,p,sizeof(struct foo));
}
```

The alternative of copying a `struct` field by field might be slower than bulk copy via `memcpy`.

Interestingly, there are many other uses of casts in C that are common, but are technically not supported by the language definition. But in any case, most compilers support them.

Unsafe code via Escape Hatches

One way that type-safe languages work around expressiveness problems is to have "escape hatches" that allow potentially unsafe operations. In OCaml, for example, you can call into C code to do your dirty work, using the OCaml Foreign Function Interface. Or, if you know enough about OCaml representations, you can even do unsafe things in OCaml itself via `Obj.magic`, part of the Obj module.

Command Injection

A type-safe language will rule out the possibility of buffer overflow exploits, and this is an important benefit. Unfortunately, type safety will not rule out all forms of attack.

In this unit we'll talk about **command injection** as another form of code injection attack, but one that requires a bit more programmer vigilance to prevent.

What's Wrong with this Ruby Code?

The following is the program `catwrapper.rb`, written in Ruby.

```
1  if ARGV.length < 1 then
2    puts "required argument: textfile path"
3    exit 1
4  end
5
6  # call cat command on given argument
7  system("cat "+ARGV[0])
8
9  exit 0
```

The intention of this program is simply to call the system `cat` command to print out the contents of the argument that was passed to the program (line 7). If no argument is passed, the program prints an error message and exits (lines 1-4).

Here's an interactive shell session, running the program:

```
> ls
catwrapper.rb
hello.txt

> ruby catwrapper.rb hello.txt
Hello world!

```

```
 8  > ruby catwrapper.rb catwrapper.rb
 9  if ARGV.length < 1 then
10  puts "required argument: textfile path"
11  …
12
13  > ruby catwrapper.rb "hello.txt; rm hello.txt"
14  Hello world!
15
16  > ls
17  catwrapper.rb
```

First, we call `ls` to list the contents of the current working directory (lines 1-3). We see the `catwrapper.rb` program itself, and a file `hello.txt`. Then we execute `ruby catwrapper.rb hello.txt` – this runs the `catwrapper.rb` program with `hello.txt` as its argument, and the result is that `cat hello.txt` is called, which prints the contents of that file (lines 5-6). Then we call `catwrapper.rb` on itself, which causes its own code to be printed (lines 8-11). The next command is the most interesting: the argument to `catwrapper.rb` is `“hello.txt; rm hello.txt”`, which looks unusual. The program doesn't reject this argument and prints out the contents of `hello.txt` as before. But on line 16 when we `ls` the directory we see that `hello.txt` is no longer there. What happened?

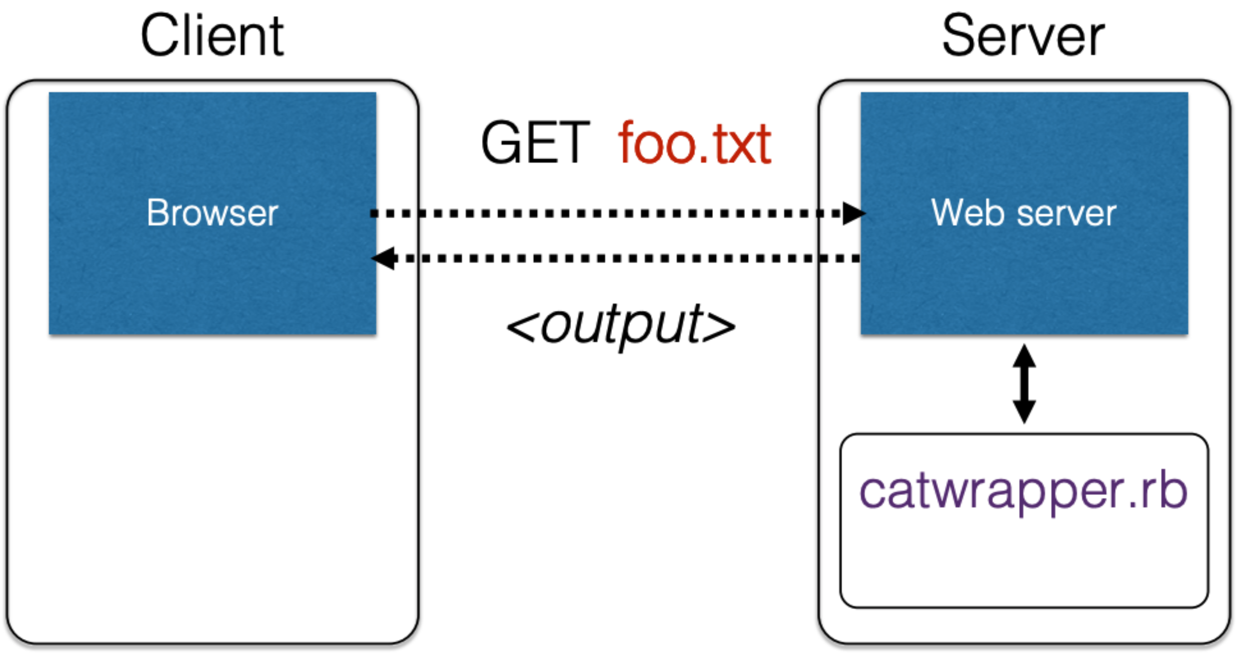
The Answer

What happened is that the Ruby `system()` command (line 7 in `catwrapper.rb`) interpreted the string `ARGV[0]` as having two commands, and executed them both! That is, when calling `catwrapper.rb` with `"hello.txt; rm hello.txt"` we end up invoking `system("cat hello.txt; rm hello.txt")`. This means that we will `cat` (print) the contents of `hello.txt` but then remove it!

This was likely not our intention in writing `catwrapper.rb`. Instead, we were simply interested in printing the contents of a file, passed as an argument. Rather than implement this functionality from scratch we used the existing `cat` program. This is all well and good. But the way we invoked `cat`, via `system()`, allowed for problems that we didn't appreciate.

When could this be bad?

If `catwrapper.rb` is just a program that we use in our own work, then it's not a big deal. We are not going to pass it weird arguments. But suppose that `catwrapper.rb` ends up getting incorporated into a web service, e.g., in a Ruby on Rails web application. Perhaps a client browser can send a request to the server and this request is translated directly into an argument to `catwrapper.rb`, whose output is then sent back to the client.



catwrapper.rb as a web service

Now a malicious user in a far-flung place in the world could send the input `"hello.txt; rm hello.txt"` and thereby corrupt the filesystem on the web server.

The problem is that the input given to `catwrapper.rb` in the web server context is *untrusted;* we cannot assume well-meaning clients, so we must allow for the possibility that the input could be absolutely anything. If we want clients only to be able to see the contents of files, then the current code is too powerful. The fix? We need to add code to **validate the inputs**.

Command Injection is Code Injection

As with the stack smashing attack we discussed in the context of buffer overflows, command injection is a kind of code injection attack. In both cases, the program expects to receive and process data, but the adversary is able to trick the program to treat received data as code. For stack smashing, this is machine code. For command injection, it is shell commands.

Defense: Input Validation

Summarizing what we have learned so far, there are two core elements that make a command injection exploit a real threat:

- There are possible, though atypical, inputs that could cause our program to do something illegal
- Such atypical inputs are more likely when an untrusted adversary is providing them

The same two elements are present for buffer overflow exploits, too. With the Morris worm, the `fingerd` program would accept any string as input, but strings over 500 characters in length would overrun the input buffer, and particular choices of string could induce (very) bad behavior. Such inputs would be atypical when coming from a normal (trusted) user, but were possible (and happened!) when coming from an adversary.

A type-safe language makes the problematic inputs for a buffer overflow impossible: Any attempt to go outside the buffer is blocked by the language. But with command injection the language does not help us. So, what can we do? We must add code to **validate the inputs**.

Checking and Sanitization

Input validation refers to a process of making an input *valid for use*, meaning that it cannot possibly result in harm if provided to our service. A valid input will be a subset of all possible inputs that a user (including a malicious one) could provide. For our `catwrapper.rb` example, valid inputs are, ultimately, names of files whose contents a particular user is allowed to see. Since `"hello.txt; rm hello.txt"` is not the name of a valid file, it is not a valid input.

There are two general ways we can write a program to ensure inputs to it are valid:

- We can *check* that inputs match the desired form. If they do not then they are rejected and the program proceeds no further.

- We can force inputs to match the desired form by modifying them, if necessary, before proceeding. Modifying a potentially invalid input so it becomes a valid one is called *sanitization*.

There are three kinds of input validation in common use: **blacklisting**, **escaping**, and **whitelisting**.

Blacklisting

Blacklisting works by preventing inputs that include features that could make it dangerous. Blacklisting has variants involving either checking or sanitization.

Checking

For example, we could replace the call `system("cat "+ARGV[0])` in our `catwrapper.rb` program to be the following.

```ruby
if ARGV[0] =~ /;/ then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end
```

The first line checks the argument passed to the program `ARGV[0]` against a regular expression that matches any string that has a semicolon (`;`) in it. If the argument matches the regular expression it is rejected, which blocks the attack that we saw previously.

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
illegal argument
```

Sanitization

Rather than checking and rejecting a problematic input, we could sanitize the input to conform to our well-formedness condition. So we could replace the above 6-line code fragment that rejects those inputs containing a semicolon with the following line of code, which simply removes any semicolons present.

```
system("cat "+ARGV[0].tr(";",""))
```

If we run this version of `catwrapper.rb` with our problematic input, we get the following.

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
Hello world!
cat: rm: No such file or directory
Hello world!

> ls hello.txt
hello.txt
```

Since the semicolon from the input is removed, the call to to `system` ends up being `system("cat hello.txt rm hello.txt")`. When the `cat` program is given multiple arguments, it prints them out in sequence. Here, `cat` thinks it is being asked to print three files, `hello.txt`, `rm`, and `hello.txt` again. The first and third arguments name a valid file in the current directory, so its contents are printed (twice). But `rm` is not a file in the current directory, so `cat` complains: `rm: No such file or directory`.

This example illustrates that while sanitization may permit accepting more inputs, one drawback is that bogus inputs can lead to confusing error messages going back to the user.

Escaping

Escaping is a kind of input sanitization that replaces potentially problematic input characters with benign ones (rather than, say, removing them, as with blacklisting). Just as with the checking variant of blacklisting, we can specify problematic characters to find and replace using a regular expression. Here's a replacement for the call `system("cat "+ARGV[0])` in our original `catwrapper.rb` program that does escaping first.

```ruby
def escape_chars(string)
  pat = /(\'|\"|\.|\*|\/|\-|\\|;|\||\s)/
  string.gsub(pat){ |match| "\\" + match }
end
system("cat "+escape_chars(ARGV[0]))
```

The `escape_chars` function takes argument `string` and uses the `gsub` function to replace substring matches of regular expression `pat` with the matched substring `match` preceded by a double-backslash (`"\\"`). For example:

- `escape_chars("hello")` returns `"hello"`
- `escape_chars("hello there")` returns `"hello\ there"`
- `escape_chars("rm -rf ../")` returns `"rm\ \-rf\ \.\.\/\"`

Running our modified `catwrapper.rb` program with the original exploit, we get the following.

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
cat: hello.txt; rm hello.txt: No such file or directory

> ls hello.txt
hello.txt
```

What happened? The `system` command ends up being called with the string `"cat hello\\.txt\\;\\ rm\\ hello\\.txt"` . The `system` command interprets this string as a shell command with escaping applied, so that it treats the sequence `\\` as an escaped backslash. It therefore strips each leading backslash so that what is passed to the shell to be executed is `"cat hello\.txt\;\ rm\ hello\.txt"` . To the shell, `hello\.txt\;\ rm\ hello\.txt` is interpreted as a single argument to `cat` , since all spaces are preceded by a backslash. So `cat` treats the argument as a file whose name has spaces in it. The error message above indicates that this file is not found, as expected.

Escaping is Language Specific

In our example we are creating a shell program to pass to the `system` command, which interprets that program. We are escaping characters that might otherwise be treated as code by the `system` command and the shell interpreter to be treated as data instead. What characters may be mistaken as code is language specific, so there is not a one-size-fits-all escaping process.

There are standard escaping functions on a per-language basis. For example, Ruby's `CGI::escape(str)` function will escape `str` so it can safely appear in a URI, e.g., `CGI:escape("'Stop!' said Fred")` becomes `"%27Stop%21%27+said+Fred"`.We will see later on the notion of a *prepared statement* for SQL queries; SQL is a programming language for accessing the contents of a database. In essence, the process of putting together a prepared statement is tantamount to escaping user input so that it is not misinterpreted as SQL code.

Whitelisting

Whitelisting is a form of checking—rather than checking that bad characters are *not* present, it confirms that only *good* characters are.

Motivation

Why use whitelisting? One challenge with both blacklisting and escaping is making sure that you identified all of the dangerous characters. In our example `escape_chars` function, for example, we left out `&` from the `pat` regular expression. Since this character is not escaped, if provided in the input to `catwrapper.rb` it will be interpreted as code, directing the shell to "run in the background."

Escaping also has the problem that it doesn't always render command characters harmless. For example, suppose in the following we use the form of `catwrapper.rb` with our `escape_chars` function.

```
> ls ../passwd.txt
passwd.txt

```

```
> ruby catwrapper.rb "../passwd.txt"
bob:apassword
alice:anotherpassword
```

Despite escaping the `.` and `/` characters, `catwrapper.rb` still dutifully follows the path to directory one level up from where it is run and prints the contents fo the `passwd.txt` file.

A web service for `catwrapper.rb` probably only intends to give access to the files in the current directory; the `../` sequence should have been disallowed, but escaping was not up to the task of doing so. A failure to disallow it is called a **Path Traversal** vulnerability. Using blacklisting to filter out `..` sequences is one solution. (But, perhaps filtering out `..` would unnecessarily disallows paths like `foo/../hello.txt` where `foo` is a subdirectory in the current working directory.)

Making Sure Input is Good (not Bad)

Whitelisting checks that the input is *known to be safe* rather than just known to be unsafe. For `catwrapper.rb`, we only want those files that exactly match a filename in the current directory.

```ruby
files = Dir.entries(".").reject {|f| File.directory?(f) }
if not (files.member? ARGV[0]) then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end
```

The above code reject inputs that do not mention a legal file name.

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
illegal argument
```

The rationale for whitelisting is that given an input, it is safest to reject it unless we have evidence that it's OK. This is the principle of *fail-safe defaults*. Attempts to just reject known-bad inputs may miss ones we don't know about, and attempts to fix known-bad inputs may produce inputs that result in wrong output, or even permit exploits.

Whitelisting is not a panacea. It may be expensive to compute the whitelist at runtime—getting the list of valid files is not cheap. A whitelist may also be hard or impossible to describe (e.g., “all possible proper names”). But from a security perspective a whitelist is preferred when possible, since it's the safest alternative.

Summary

To sum up: Input validation is needed when not all of the *possible* inputs to a piece of code are sure to be *safe* inputs.

Any inputs arriving from the outside world, which could be from an untrusted or adversarial party, should be treated with suspicion. Any code that directly consumes such inputs should be scrutinized, as should code that subsequently processes outside inputs. The *taint* of adversarial influence could, without care, propagate through a system to files, databases, etc. and subsequently harm future interactions.

Input validation—in the form of blacklisting, escaping, and whitelisting—can make sure that untrusted input is made safe.

WWW Security

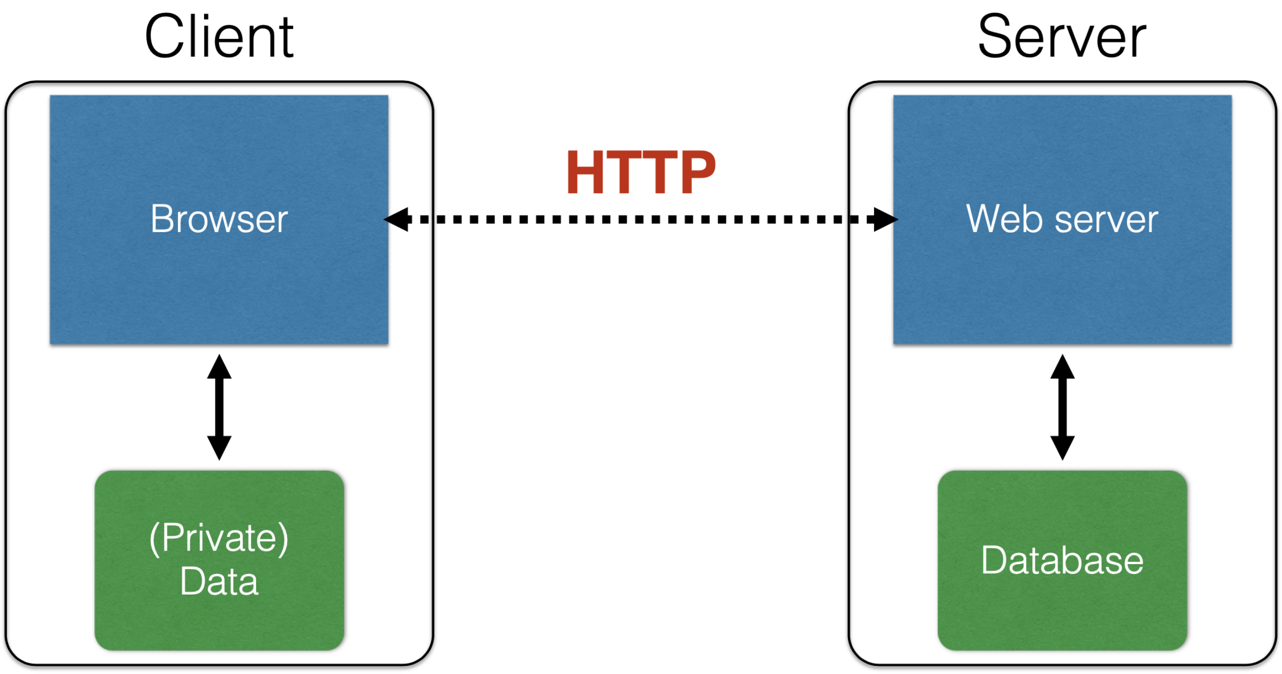
Much of what the world does today with computers involves communications via the Internet.

Much of what happens over the Internet is driven the World-Wide Web (**WWW**), or simply "*the Web,*" which is defined by exchanges of *HyperText Markup Language* (**HTML**) documents (and many others besides) between clients and servers via the *HyperText Transfer Protocol*, or **HTTP**, and its cryptographically secured version, **HTTPS**. Clients and servers both have resources that need protecting, including personal records, medical information, corporate intellectual property, bank account balances, stock holdings, and more. In short, *many of the things we value are accessible via the Web.*

Given the rich trove of valuable assets on Web-connected computers, attackers are constantly trying to subvert protections of those assets. One way they do this is to find and exploit vulnerabilities in software running on Web-connected computers, including web browsers, web servers, database management systems, and operating systems. This constant threat means that programmers of Web-accessible software need to be careful not to introduce vulnerabilities in their code. Even code not originally written for use over the Web often ends up there, so not intending to write for the Web is no excuse. Essentially all programmers are on the front lines.

In this unit, we will first introduce the basics of the Web – who the players are and how things work together. Then we are going to look at two common, dangerous WWW attacks: **SQL Injection**, and **Cross-Site Scripting**. Both are a kind of **code injection** which works by getting a program to confuse what it intends to be data as code. As it turns out, we can defend against them by employing **input validation** techniques, just as we previously discussed for defending against buffer overflows and command injection attacks. One new challenge is that the Web's ubiquitous use of **mobile code**, particularly Javascript programs.

WWW: The Basics



The basic structure of web traffic

In the simplest terms, the WWW consists of clients and servers. The servers host content (aka *resources*) that clients would like to access. Clients send requests, often using a browser employing the web protocol HTTP, to web servers that receive and process those requests. The remote server may store content within a relational database, on its local filesystem; requests by clients are translated by the web server into requests for this local content. As it turns out, a server may store very little content itself; in this case, the web server basically works by translating its clients' requests into requests to other servers, and then translates and relays the responses back. As mentioned above, both clients and servers have sensitive content that needs to be protected from attack.

Universal Resource Locators (URLs)

Resources accessible via the WWW identified by a URL, or *Universal Resource Locator*. A URL consists of three parts: The *protocol*, the *hostname/server*, and the *path*. For example, in the URL `http://www.cs.umd.edu/~mwh/index.html` , the protocol is `http` ; the hostname is `www.cs.umd.edu` ; and the path is `~mwh/index.html` . Here's a little more information about each of these.

- The **protocol** portion of a URL identifies the communication protocol by which that the remote server will return the specified resource. For our example, `http` identifies the standard HTTP, which we'll say more about below. Other examples include `ftp` (for *file transfer protocol*), `https` (for *HTTP secure*), and `tor` (which employs *onion routing* to help make the client anonymous to the server).

- **Hostname**/server is a name of the remote server that serves the specified resource. This name must be translatable to an Internet Protocol (IP) address via the Domain Name Service (DNS). In addition to names like `www.cs.umd.edu` or `facebook.com`, IP addresses like `128.8.127.3` can also be used.
- The **path** is the "address" of the resource at the remote server. In the simplest case, this path is translates pretty directly to a file on the remote server's filesystem. For example, `~mwh/index.html` might translate to `/fs/www/users/mwh/index.html` on the filesystem of the machine servicing requests to `www.cs.umd.edu`. On the other hand, a path is often more complicated. For example, the path `delete.php?f=joe123&w=16` indicates that the remote resource is file `delete.php` which contains code that should be run with arguments `f=joe123` and `w=16`. As such, the path identifies *dynamic content*; it's a PHP program that will be executed at the server with the given arguments, returning content so that it can be viewed at the requesting client.

HyperText Transfer Protocol (HTTP)

A typical WWW interaction starts with a client, perhaps because a user clicks on a web page rendered in their browser. This click results in an **HTTP request** being made for a resource named by an associated URL. Maybe the click was on a thumbnail picture, and the request amounts to requesting the full-sized picture from the remote server.

An HTTP request contains several elements, most notably the **URL of the resource** the client wishes to obtain and the **request method**; the two most common methods (covered by the vast majority of Web traffic) are `GET` and `POST`. (Other request methods include `HEAD`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`, and `PATCH`.) The `GET` method indicates the request to retrieve (read) data associated with the given URL, with no changes being made to any server-hosted data. The `PUT` method indicates that the request includes data being provided by the client, e.g., as part of a web form, and as such the server might update its local state.

A request also contains various **headers** which provide information about the browser (e.g., which browser and version is being used), local capabilities (e.g., whether the requesting machine is full-scale computer or a small mobile device), and resources the client may have that are relevant to the request (e.g., cookies, which we will discuss shortly).

Now we will consider the two main request methods, and what responses to requests look like.

HTTP GET

An HTTP request is a textual message formatted in a certain way, according to the protocol specification. Suppose we type in `http://www.reddit.com/r/security` into our browser. This will result in a text message like the following being sent to `www.reddit.com` :

```
GET /r/security HTTP/1.1
Host: www.reddit.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11)
Accept: text/html,application/xhtml+xml,application/xml;...
Accept-Language: en-us,en;q=0.5
...
```

We see in the first line the request method, `GET` , followed by the URL path `/r/security` , followed by the protocol `HTTP` (version 1.1). On the next line is the server/hostname, `www.reddit.com` . The remaining lines contain the request's headers. The `User-Agent` header indicates the software making the request; it is typically a browser, but it could be something like `wget` , the JDK, etc.

We will consider some particular headers in more depth, shortly. One header of interest is the referrer URL. This header is contains the name of the resource that linked to the resource being requested. For example, if we visited a page at `/r/security` and clicked a link, we might get a request like this:

```
GET /worst-ddos-attack-of-all-time-7000026330/ HTTP/1.1
Host: www.zdnet.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11)
...
Referer: http://www.reddit.com/r/security
```

The (incorrectly spelled) referrer field indicates that the request to `www.zdnet.com` came from a link on a page hosted at `www.reddit.com`. This information turns out to be useful to defending against a certain sort of attack, called **cross-site request forgery**. But, more on that later.

HTTP POST

An HTTP POST request often arises when a user fills out a form on a web page. When that user clicks submit, the POST request is sent. The final part of the request contains the client-supplied content. The following is an example of a request made when submitting a post to `www.piazza.com`.

```
POST /logic/api?method=content.create&aid=hrteve7t83et HTTP/1.1
Host: piazza.com
User-Agent: Mozilla/5.0 ...
...
Cache-Control: no-cache
  {"method":"content.create",
   "params":
     {"cid":"hrpng9q2nndos",
      "subject":"<p>Interesting.. perhaps it has to do ..."}}
```

The last part of the request in (in curly brackets) names the submitted fields ("keys") and their corresponding values. For example, this POST sets the key `method` 's value to `content.create` and the key `params` ' value to be a list of other key-value pairs. (This data is encoded according to the format indicated in the header `Content-Type`; a typical choice is `application/x-www-form-urlencoded` which is harder to read than what's shown above.)

Note that these days it's often the case that data is supplied not just in the POST body but also as parameters in the URL itself, as indicated in our discussion of URLs, above. Indeed, we see the use of parameters in the `piazza.com` POST request, in the form of the `&aid=` ... part of the path. However, the intention here is that URL parameters are meant to simply name a resource, which has no effect on the server's state, whereas the POST body's content is the basis for an update to the server's state.

HTTP Responses

After processing the request from the client, the server sends back a response. For our example `www.zdnet.com` GET request, here's a possible response.

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=abc123;path=/;domain=zdnet.com
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Content-Type: text/html; charset=UTF-8

<html>This is the body of the <b>response!</b></html>
```

The first line indicates the *protocol* being used (`HTTP/1.1` in this case), followed by the *status code* (`200` in this case), followed by the *reason phrase* (`OK` in this case). The lines that follow which begin with a name followed by a colon are the headers (e.g., `Date:` , `Server:` , etc.). The `Set-Cookie` headers are of particular note, as they tell the browser about server-side state that the browser should store on the server's behalf; we'll discuss these in more depth later. The very end of the response contains its content. In this case, as indicated by the `Content-Type` header, it is formatted as HTML.