CS 155 NOTES

ARUN DEBRAY
APRIL 23, 2015

These notes were taken in Stanford's CS 155 class in Spring 2015, taught by Dan Boneh and John Mitchell. I TeXed these notes up using `vim`, and as such there may be typos; please send questions, comments, complaints, and corrections to debray@cs.stanford.edu.

CONTENTS

Part 1. Introduction and Overview

Lecture 1.

Introduction, and Why Security is a Problem: 3/31/15

"What you would do, is you would take over — well, not you, but..."

There are no textbooks for this class; the material moves very quickly, and any kind of textbook would quickly become out of date. Instead, our readings will be from papers posted on the website. Additionally, the first programming project is up, though it won't even be explained until Thursday.

Let's start by talking about computer security on a high level. There's lots of buggy software and exploitable systems on the Internet. No single person can keep the software of an entire system in their head, and therefore the boundaries of code written by different people commonly causes these vulnerabilities. In addition to buggy software, social engineering, e.g. phishing, leads people to install software that perhaps they would prefer not to install.

Moreover, finding security vulnerabilities can be incredibly profitable, not just from the opportunties of taking over computers (in many ways) but also discovering vulnerabilities and selling them to others.

This combination of many potential vulnerabilities and very profitable vulnerabilities leads to their importance, and why we're all sitting in a class about it.

Moreover, vulnerabilities are getting more and more common as more and more people and devices use the Internet. About 45% of the vulnerabilities exploit Oracle Java (oops), since the JVM isn't the most secure code in the world and is installed in many browsers. The professor recommends turning off the JVM in your browser. 42% of these vulnerabilities come from browsers in other ways; other remaining common exploits (1% to 5%) come from Adobe Reader (so malicious PDFs!), Adobe Flash Player, Microsoft Office, and the Android OS.

Another large recent trend is mobile malware; for example, banking trojans grew immensely in 2014. These are pieces of software which imitate a bank website or app and record the account information and password for less noble purposes.

Here are some examples. Why would you want to own a personal computer?

Example 1.1. The professor's parents run a Windows machine, to his chagrin: every other week they get some sort of malware. Why do all these people want to exploit that machine? It doesn't have anything that scandalous on it — but one can use it for all sorts of things.

For example, one could steal its IP address; now, it looks like a random Internet address, so it's a nice place to send spam from. It'll get blocked fairly quickly, but until then plenty of spam can be created. There are economic studies of the business of spam — so-called "spamalytics" — one email in 12 millon (i.e. *very* close to zero) leads to a purchase. But since sending spam is basically free ($\ll$ one cent), the spammer is happy. Relatedly, one greeting card in 260,000 results in an infection.

Another use of personal computers is as part of botnets which can be bundled up into services. For example, one can group together many personal computers to participate in distributed denial of service (DDoS) attacks, which cost on the order of $20 an hour.

Yet another use is click fraud; in advertising on the Internet, one earns money per clicks on an ad. However, one could make a bot on someone else's computer which keeps clcking on advertisements. This is fraud, and there's a lot of work in either direction trying to create or combat this.

Just by the way: **the material in this class is for educational purposes. Do NOT do this at home; it is pretty illegal.**

Another use of someone else's machine is to steal their credentials, e.g. by using a keylogger. This could gather one's banking or web passwords, and so on. Alternatively, one could inject one's own ads and make money that way. A good example of this was the SuperFish proxy which spread around fairly recently; we will discuss it further when we discuss web security, later in the class. One way credential stealing can work by injecting Javascript into a login page that forwards the user's inputs to another site.

A third reason to own one's computer — remember Stuxnet? The trick was that the system they wanted to gain access to was isolated from the Internet, because it was important to keep it isolated. So the perpetrators infected a bunch of random laptops with the hope that one would connect to the isolated system, and then when this happens, come out of dormancy and do Evil Stuff.[1]

One can also infect servers. Why is that? Servers are treasure troves of data: credit card data, user information, etc. One of the classic examples is from 2013 on Target, which retrieved about 140,000,000 credit card numbers! The industry took quite a while to recover from that; it takes quite some time and effort and expense to revoke those credit cards. These kinds of attacks are rampant, and have been since the beginning of the millenium.[2]

Server-side attacks can be politically motivated, too. Even right now, there was a server-side attack on Baidu where people searching there also generate requests to Github. We don't know who is doing this or why, but it's related to political Github projects, and therefore is politically motivated.

Another "good" reason to infect a server is to infect the users of the website. There's a piece of software called MPack which makes this relatively streamlined, and has been installed on 500,000 websites and countless end-user machines. This is a pretty effective use of a large web server.

A defense mechanism to this is Google Safe Browsing, which blocks sites that it detects with malware; Chrome, Firefox, et al. then refuse to go to those sites.

Insider attacks are created by people who are part of the project. For example, there was one in the Linux kernel a few years ago that "accidentally" gave the user root access. Sometimes, these happen in companies by disgruntled workers, etc. It's particularly hard to detect insider attacks programatically, and the very cleverest ones can be made to look like mistakes (c.f. the Underhanded C Contest).

Now, suppose you've found a vulnerability in code. What can you do now? One option is a bug bounty program; for example, Google has a standard program for reporting bugs and paying the discoverers. The bounty can be quite nice; some people make a living doing this. Microsoft and Mozilla have similar programs.

---

[1] Conversation between the professor and one of the tech people: "You guys killed my computer, so that's nice."
"Yeah, I'm not sure what's going on."
"That's exactly the kind of thing I was hoping not to hear."

[2] "We're back online… or not."

These are examples of *responsible disclosure*, which includes staying quiet about it until the patch is released. One can also sell these exploits to antivirus vendors.

Of course, you (well, one) can also sell it to the black market. These also pay well. But who buys? Use your imagination.

Then, what happens once it's found? There are whole industries to take advantage of them. For example, pay-to-install services (PPI) use these exploits to take over machines, and then the customer's software gets run. Surprisingly, the cost of infecting, say, a thousand machines in America is only $100, and it's cheaper in Asia (about $7!).

In this class, we're going to learn a lot about these things, with the goal of being able to know what the threats are and design code or systems that are safer against these attacks. This includes program security, web security, network security, and mobile app security.

Once again, never ever ever try this yourself.[3] First of all, Stanford's security team has no sense of humor at all. Neither does the law. People take this very seriously.

Ken Thompson's Clever Trojan. The question is, at its essence, *what code can you trust?*

In Linux there are commands like `login` or `su` that require one to enter one's password, or more generally credentials. How do you know that these are safe and won't send your credentials to another person? Do you trust the Ubuntu distribution?

Well, if not, let's download the code and inspect the source code. We can be reasonably confident that the code is safe, then, and then compile it and get an executable.

Ah, but can we trust the compiler? This is a real question: it's perfectly possible for a compiler to be made malicious, in which it checks if the program is the `login` program and adds some extra code to add the backdoor and compromise your password. Otherwise, it can act as a regular compiler.

Great, so now we need to look at the compiler's source. Then we can compile it — but we're still dependent on the compiler. A C compiler is written in C these days, so consider an attack which does the following: it changes the compiler's source code to the malicious compiler from before, which makes the `login` program with a backdoor and is otherwise secure. But additionally, if it sees *the compiler's* code, it replaces the compiler with its own code, so the regular compiler is built into the malicious one! (Otherwise, it compiles normally.)

Thus, we compile the compiler and end up with the malicious compiler, so `login` is still compromised. Oops.

The compiler back-door has to print out the code for detecting the compiler source code, which is rather nontrivial, since this code contains the backdoor. Thus, it has to be a quine — it has to print out its own code!

Ok, let's go deeper. Now, the attacker removes the malicious source code, because the executable is still malicious, and will poison all compilers that the user tries to compile. The source code is pristine, and secure… but there's nothing we can do.

So the user has one option: to write the `login` program in assembly. And the entire compiler. Yuck. This is extremely impractical for any real software. But not just text-based assembly: the assembler could be malicious, so you have to write the binary yourself.

In other words, if we don't trust anything, we're going nowhere fast, so we have to have some root of trust that we can base the security on. Well, we can't really trust the compiler, and similarly the OS could be malicious too! We shouldn't trust the hardware, either… but at this point, we have no choice, even if we reduce to just trusting the `x86` processor. So, better hope Intel doesn't go rouge. There is a lot of work towards reducing how much trust we put in our hardware, e.g. graphics cards, network cards. Perhaps only the `x86` processor is the root of trust; that would be all right. This is hard, but would be possible.

Lecture 2.

Control Hijacking Attacks: 4/2/15

"The graph stops at 1996, I guess because I got tired of typing in numbers."

Control hijacking attacks are a very common class of attacks, including buffer overflow attacks, integer overflow attacks, and format string vulnerabilities. These involve using mistakes in a program to gain control of a running program or process.

[3] Like, ever.

Buffer overflows are very old vulnerabilities, dating back to 1988. Hundreds are discovered every year, and they are an active area of research. These mostly affect programs written in C and C++.

To understand these, you need to understand a bit about system calls, including the `exec()` system calls, and the stack and heap structure. `exec()` takes a string and then runs the program named by that string in the current address space. This does require knowing what CPU and OS the computer is running, but this is very easy, e.g. connect to a web server or using specialized software. In this class, though, we'll stick with `x86` CPUs running Linux or Windows. There are also slight differences between CPUs, e.g. little-endian or big-endian architecture.

Recall the memory map for a typical Linux process, e.g. a web server. The bottom memory is unused; then, the program code is on top of that, loaded from the executable. Then, the runtime heap comes after that, which grows upward, followed by shared libraries and then, at the very top, the user stack, which grows downwards, and is delineated by the extended stack pointer `%esp`.

When one function calls another, another stack frame is created. The following are pushed onto the stack, in order (so, growing downward):

- The arguments to the function.
- The return address of the callee (pointing back to where we were in the caller).
- The stack frame pointer, which points to the previous stack frame, useful for debugging.
- In Windows, exception handlers are pushed next. These are ripe vectors for attacks.
- Next, local variables are pushed.
- Finally, the callee's saved registers are pushed.

This should be familiar to you by this point.

Now, suppose we have a really dumb function in this webserver, as follows.

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do_something(buf);
}
```

So now, we have 128 bytes of space just below the stack frame pointer. Thus, if the buffer reads more than 128 bytes (specifically, 136 bytes), the return address and stack frame pointer will be overwritten. In particular, the return address can be modified to another address, so a hacker can cause the program to pass to any other point and execute arbitrary code. This is why it's called control flow hijacking.

So how does this turn into an exploit? The hacker submits much more than 136 bytes; in addition to a bogus return address, it should also contain the code for some bogus program P, e.g. `exec("/bin/sh")`. The return address should point into the address where P is loaded, so that when this function returns, it returns right to the beginning of P, and therefore begin executing a shell. Now, the attacker has a shell on the system with the priviledges of the web server and can do what he wants. This is often game over, e.g. the attacker can run its own programs, change websites, and so on.[4]

In practice, it's not just shell; there's some bookkeeping to do, setting `stdin` and `stdout`. This is called a *shell code* more generally. There's no need to write this yourself; there are famous well-known ones, e.g. the one we'll use, by someone called $\aleph_1$. The point is to format this into a bogus URL, which also contains P and the return address.

Now, there's one more ingredient missing: how does the attacker know what the address of P is? If he fills it with a random address, the server will probably just crash. Poor hacker! What does he do? The attack string must be modified in the following way: in addition to the 128 bytes of the buffer, the attacker provides an *approximate* return address (it's possible to get good estimates by running the web server on one's own machine), which points into a 10,000-byte (or so) sequence of `nop` instructions, called the *NOP slide*. Each instruction does nothing, and so it slides into program P (which is placed above the NOP slide, i.e. after it in the string). The idea is that as long as the code reaches the NOP slide, it doesn't actually matter where it goes.

Finally, how does the hacker know that the buffer is 128 bytes? This can come from open-source code, or by getting a copy of the web server and playing with it until a vulnerability is found.

[4]Next week, we'll talk about systems that remain secure even when some of their components are compromised.

Buffer overflows are extremely common, and very difficult to prevent; we're just human, and static analysis has bugs too. Of course, not using `strcpy` is a good idea, but more common is to structure code to weaken buffer overflows, as opposed to preventing buffer overflows. For example, one might try to prevent the processor from executing code on the stack.

There are some tricks that make this a little harder. For example:

- Program P cannot contain the null character. Programming without zero is like writing English without the letter e; $\aleph_1$ has already solved this problem, so don't worry about it, but it's an interesting thing to think about. Moreover, since this is a significant restriction on shell codes, one might try to scan for $\aleph_1$'s shell code... but this is pretty hopeless too. There are still lots of shell codes and some of them can be very very obfuscated, e.g. one shell code that looks like English poetry in ASCII. Similarly, the NOP slide can be made to look like reasonable English text. Some shell codes even work on multiple processors.
- The overflow should not cause the program to crash before `func()` exits, of course.

There are some well-known remote stack smashing overflows.

- In 2007, there was an overflow in Windows animated cursors, in the function `LoadAniIcon()`. Even though IE was compiled to avoid these issues, this overflow was so optimized as to be undetected, so that if one goes to a website or opens an email containing a specific animated GIF, the adversary gets shell on the user's browser.
- In 2005, there was an overflow in Symantec's antivirus software (huh, that's pretty ironic); from the browser, one could send some Javascript to run code with the privileges of the anti-virus system (which is typically admin!).

This is a very common problem; many of the functions in the C standard library are unsafe, e.g. `strcpy`, `strcat`, `gets`, and `scanf`. There are "safe" versions, e.g. `strncpy` and `strncat`, which copy only n characters (n supplied by the programmer). However, these don't always leave their strings terminated! Oops. This can cause overly long strings to crash the program anyways, and thus are unsafe. In the Windows C runtime, the safer versions, which do terminate their strings, are called `strcpy_s`, etc.

There are plenty of other places one can find opportunities for a buffer overflow, e.g. on Windows, one can overwrite the addresses of exception handlers in the stack frame, and then invoke an exception. Function pointers are also useful; for example, if a buffer sits right next to a function pointer, then overflowing the buffer can allow someone to change the value of the function pointer. This is harder in practice, not just because of the NOP slide and all that, but also because it involves guessing where things are on the heap, which is difficult and changes. However, there's lots of techniques both to attack (heap spraying, where the heap is filled with vulnerable function pointers and shell code) and defend against these.

This is particularly concerning in C++, because it has methods, i.e. basically compiler-generated function pointers. If the table of function pointers (an object's method) is placed next to a buffer, the result is a buffer overflow — slide in a shell code and you're set.

Finding Buffer Overflows. To find an overflow, one can run a web server on a local machine, and issue it lots of malformed requests. Of course, there are automated tools, called "fuzzers," which do this. Then, if one of the requests causes the server to crash, the hacker can search through the core dump to look at the state of memory to find the malformed request, which was the location of the buffer overflow. It's possible, but much, much harder, to look through the binary code, and therefore explicitly construct exploits, but in practice these fuzzers are the most common and powerful way to make this work. There's a lot of good engineering involved in making a good fuzzer, and a lot of companies should do this but don't. Of course, the big companies do, and thus catch some of their own vulnerabilities. Keep this in mind, and fuzz your code!

More Kinds of Control Hijacking Attacks. There are many different kinds of control-flow hijacking attacks; we just saw stack-based ones, but there are also integer overflow attacks and double free attacks, which can cause the memory manager's data structure to be messed up and write data to locations that can be discovered. Another related attack is use-after-free, where memory is used after it was supposed to be free. This is very useful, because it means memory is used in two ways, so, for example, a function pointer can be overwritten with something that an adversary wants to point to. It turns out browsers are very vulnerable to this, because of DOM management; last year, this was the largest source of attacks on browsers. Finally, there are format string overflows. These are things which C and C++ developers need to be aware of, and often are not.

Returning to integer overflows, recall that `int`s are 32 bits, `short`s are 16 bits, and `char`s are 8 bits. If one adds two things that would go outside those bounds, it causes an overflow. Can this be exploited? Yes! Consider the following code.

```
void func(char *buf1, *buf2, unsigned int len1, len2) {
    chart temp[256];
    if (len1 + len2 > 256) { return -1;} // length check
    memcpy(temp, buf1, len1); // cat buffers
    memcpy(temp+len1, buf2, len2);
    do_something(temp); // do stuff
}
```

So imagine that the attacker can make `len1 = 0x80` and `len2 = 0xffffff80`? Then, their sum is zero! Then, `memcpy` will copy an enormous number of bytes into the heap; since the heap grows upwards towards the stack, this will cause things to be written to the stack, at which point the attack continues as before.

How could one check against this?

- If the sum of two positive numbers is less than either number, there is a problem. This is a good way to check this in the above code.
- Alternatively, for signed integers, one can check that their sum is not negative, which indicates an overflow.

Finally, let's talk about format string issues, which will pop up on the first project. An inexperienced developer might write the following program.

```
void func(char* user) { // i.e. this string was supplied by the user
    fprintf(stderr, user);
}
```

Thus, if the user can supply the format string, it can peek around, since further arguments would come from the stack! Thus, one could supply `"%s%s%s%s%s%s%s%s%s%s"`, which means the user can see the memory of the program! However, the program will most likely crash, e.g. if some of the words are 0, so not really pointers.

The correct code is `fprintf(stdout, "%s", user)`. This is true for any function taking a format string; do not be lazy!

Platform Defenses. This is an active area of research: anything that works will probably be useful, so if you have any ideas, go talk to someone! Sometimes this also comes with nice prizes.

The first way to fix this problem is to avoid these bugs in the first place. There are automated tools for this, e.g. Coverity, Prefast, Prefix, and so on. These will catch some bugs, but not all of them!

Another idea is to (re)write code only in type-safe languages like Java. This is nice in theory, but there's so much legacy code that it's not always viable.

It's also possible to defend, e.g. prefer crashing to executing arbitrary code. Then, the worst a hacker can do is a DDoS.

For example, one can mark memory as writable or executable, but not both. This is called W^X ("W xor X"), which was implemented in Intel and AMD processors a decade ago. Basically, in every page table entry, there's a bit indicating whether it can be written or executed (called the NX or XD bit), and jumping execution into those pages not allowed to execute causes a segmentation fault. These are mirrored in software, with these defenses enabled in Linux and Windows. However, some programs need this ability, such as just-in-time compilers. In Visual Studio, this can be turned off, e.g. with the configuration `/NXCompat:NO`.

As a response, attackers began to figure out how to hijack control without executing code. This is the start of Return Oriented Programming (ROP). The idea is that `libc.so` contains the code for `exec()` and `printf()`, as well as the string `"/bin/sh"`. Thus, an attacker making a buffer overflow can point to the address of `exec()`, and have the argument (just above that) point to the shell string. Of course, this is a simplification, since the input and output streams have to be modified, but the key idea is in place: use code fragments which are already in memory to get a shell code to execute. Shell code basically already exists in memory!

Once again, this gets sophisticated quickly: programs can look through code for certain pieces of a script, and then construct a sequence of stack frames that make the code execute.

One defense against this is to make `exec()` (and other components of `libc`) live at different places in different invocations of the programs: the location of the library is randomized from machine to machine. Attack, defend, attack, defend, like chess or ping-pong. This method is called ASLR (address space layout randomization), where shared libraries are mapped to random locations in process memory.

Of course, randomness is hard; the first deployment in Windows 7 had only 8 bits of randomness, so one attack in about 256 succeeded! That's still pretty good. Now, however, in Windows 8, there are 24 bits of randomness, which is sufficient.

Unfortunately, this is still quite hard to get right: everything has to be randomized, even the locations of basic code in the executable! Thus, these days, when you start a program, it gets loaded into a random place, so the attacker doesn't know where to jump to. And if the attacker can figure out where `libc` is loaded (e.g., by format string vulnerabilities), then he's back in the game.

There are other ways of randomizing against this or protecting against it, but they're under research and not implemented.

Of course, the attacker is still in the picture, and can defeat ASLR cleverly, via a method called memory exhaustion.

Lecture 3.

Section 1: 4/3/15

Today's section was given by Arun Kulshreshtha.

`x86` has about six general-purpose registers, along with three special-purpose ones.

- `$esp` points to the top of the stack (lowest memory location), which is where to push or pop from. This changes a lot, e.g. with every function call or whenever anything is pushed.
- `$ebp` points to the beginning of the latest stack frame, which is where the caller's stack frame is stored. This changes a little less often.
- `$eip` points to the address of the next instruction to execute. Typically, this has a very low value, unlike the stack, which has high memory values.

When you call a function, `$eip` is pushed, `$ebp` is pushed, and then `$ebp` is set to `$esp`. This is useful because it can be unwound: to return from a function, set `$esp` to `$ebp`, and then pop `$ebp` and then `$eip` (since they're now right at the top). This is the calling convention, which we saw in CS 107. Drawing pictures about this will be very useful on the project.

Setting up the Environment. The disk images that were provided for the project can be opened with many VMs, e.g. VMWare, VirtualBox, etc. VMWare is usually not free, but it's possible to get a free student copy at `software.stanford.edu`: search for "VMWare fusion."[5] On Windows, use VM Player, which is also free.

While you can use open-source alternatives such as VirtualBox, VMWare makes it considerably easier to set it up. Note, however, that the network adapter doesn't always work, *which doesn't match all of the instructions on the sheet*; switch from bridged networking to a shared connection (with your Mac or PC). This should make it boot up faster, and work.

Then, we have a VM! It's a little weird, small font, etc. Instead of working directly, which can be painful or masochistic, you may want to `ssh` in, and you'll need to `scp` files back and forth anyways. Use `ifconfig` to get your IP address, and then `ssh` into `user@`*IP address*. If this isn't working, come to office hours or make a Piazza post.

You'll notice the machine is very empty. A good start is to download the starter code onto it (e.g. using `wget`). There's a `README`, and the Makefile makes it easier to submit; see Piazza for more instructions. The important takeaway is that this Makefile is not for building things.

Then, we get some programs, all of which are horribly buggy, and will allow us to own this machine. First, you have to build the programs, and in particular, run `sudo make install`, which will copy the programs over to the `/tmp` subdirectory, disable the stack protector, and `setuid` them to run as root. In particular, the exploits and the grading code will assume these programs run in `/tmp`.

Here's the first target program. It's pretty simple.

[5] It says "purchase," but the cost is $0.00.

```c
int bar(char* arg, char* out)
{
    strcpy(out, arg);
    return 0;
}
void foo(char* argv[])
{
    char buf[256];
    bar(argv[1], buf);
}
int main(int argc, char *argv[]) {
    foo(argv);
    return 0;
}
```

There's not much point to modifying the target; instead, the code you want to write will live in the `sploit1.c` file. Since grading will be with scripts, this will be important. This is the bare minimum of executing one program within another; you just pass in your arguments. Make sure the end of the arguments list is terminated with a null pointer.

The next ingredient is the shell code, 46 bytes long (including the null terminator), already linked in each exploit program. This was written by $\aleph_1$; you don't need to know the gritty details about how it works.

So we talked about control hijacking last lecture — there's a problem with this program! (Ominous background music) Let's give the program more than 256 bytes of input. Then, the buffer keeps going, because `strcpy` doesn't bounds-check. In particular, we can overwrite the return address of the function.

One common lesson is that unintended behavior can lead us into doing evil things.

So now, we need to figure out the mechanics of the state of the stack. This is where `gdb`, the GNU debugger, comes onstage. We can run `gdb` on the exploit, but since we're calling `execve` to run the exploit, which overwrites the address space, `gdb` gets confused.

The shell code fits comfortably within the buffer, since it's small, so we don't need to do the more elaborate method from class where it comes after the overwritten return address. Using `gdb`, we can determine what the address that we want to jump to is, as well as where the saved return address is: this is right after the buffer, flush against it.

Lecture 4.

Run-time Defenses and the Confinement Principle: 4/7/15

Today's lecture will still discuss control-flow hijacking attacks, but focus on defenses against them. For example, most applications (not JITs) can mark the stack and heap as non-executable, which stops the most basic buffer overflow attacks, but then hackers responded with return-oriented programming (ROP) attacks where hackers use existing code, rather than supply their own code. It even turns out that ROP attacks on standard programs are Turing-complete. Then, ASLR is used to make ROP attacks harder.

This is still a very active area of research, so if you have ideas, feel free to bring them to the professor.

Today's defenses, run-time checking, take the philosophy that buffe overflows and their attacks will always happen; the goal is to convert ownership attacks to crashes, since a DDoS is better than losing control. There are many of these techniques, but we'll only discuss those relevant to overflow protection.

One solution is called StackGuard, or a *stack canary*. This embeds a "canary" in every stack frame, and verifies that it's intact before a function returns; if it isn't, then it crashes. Thus, if a buffer overflows, it's likely that the wrong value will be in place, so the stack compromise is detected.

One common way to do this is to use a random value that's consistent across all stack frames; another is to use entirely null characters, which the attacker would notice, but means that attacks due to `strcpy` can't overflow the buffer: either it contains non-null characters, and the canary causes it to crash, or it does, and the string ends before the buffer is overflowed.

Since the terminator canary (the latter kind) can't protect against `memcpy` attacks, the random canary is almost universal.

StackGuard is implemented as a GCC patch (so programs must be compiled), which is now default; setting and checking the canaries causes a performance tradeoff, though at first this was about 8% on Apache and now is considerably less.

However, it isn't a perfect defense, e.g. if one is allowed to write to an arbitrary index, it's possible to jump the canary entirely, and this doesn't work for function pointer abuse or heap overflows. Some mechanisms (e.g. PointGuard) have been proposed for that, but they're less reliable, much harder to write, and come with a much greater performance overhead.

GCC's `-fstack-protector` flag, part of ProPolice, actually goes farther, rearranging the stack layout to prevent pointer overflow. The stack grows by first copying the pointer arguments, then the local non-buffer variables, then the local string buffers, then the canary. Thus, string buffers do not overflow into pointers; only into each other (which is fine) or a canary, which is detected.

Here's the details of how the check works.

```
function prolog:
    sub esp 8 // 8 bytes for cookie
    mov eax, DWORD PTR__security_cookie
    xor eax, esp // xor cookie with current esp
    mov DWORD PTR[esp+8] // save in stack

function epilog:
    mov ecx, DWORD PTR [esp+8]
    xor ecx, esp
    call @__security_check_cookie@4
    add esp, 8
```

One has to be careful; initially, these checks were only added in functions that were thought to be vulnerable, but sometimes this isn't aggressive enough; thus, now it is more aggressive.

If one has exception handlers, they're usually placed just after (above) the canary, so that the canary also protects the exception handler (which, as we saw last time, is also a source of potential vulnerabilities). In Windows, when an exception (e.g. division by zero) is fired, execution stops, and the program follows its exception handlers up the stack (linked-list) until it finds an exception handler or crashes with the famous message asking whether you want to report details to Windows (the answer, by the way, is no).

However, if the attacker overflows the canary *and* the exception handler, then manages to cause an exception, then the attakcer can use the exception handler space to load his own stuff, specifically an address pointing to code the attacker wants. The issue is that the exception handler runs off before the function returns and the canary can be checked.

Microsoft was just as surprised as anyone else by these attacks, which popped up after StackGuard and ProPolice was implemented. The first idea is to check the canary just before invoking the execution handler, but this has to be checked on every node in the linked-list (in case the attack happens and then other functions are called), so this isn't really efficient.

The solution is that the exception handler can be required to only jump to code that's marked as valid exception handlers (and crash otherwise). This was called SAFESEH. This gets rid of many attacks, but there are others that bypass this, e.g. the hacker can reroute the program into another exception. Generally, restricting the program to one of five places to go means an attacker can poke the program into jumping to one of the four wrong ones, which has been the basis of a surprising number of attacks.

Thus, the solution, called SEHOP, is to include a dummy final node at the end of the list; then, when an exception happens, it walks up the list to make sure we get to this final node and check that it's valid. It's not really possible to overwrite this stuff or get that final address, since everything is randomized. It's clever and efficient.

Of course, canaries aren't foolproof, since they don't protect against the heap or integer overflows, and some more stuff needs to be done for exception handlers. But they're really useful, and it's very important to compile code with stack canaries enabled.

But what if you don't have the source? Instead, there's a neat library called LibSafe, which is a dynamically loaded library (so there's no need to recompile the app) which intercepts calls to `strcpy`, and validates that there's enough space in the stack; if there isn't, it terminates the application. It's possible to calculate the size of the buffer as the address of the frame pointer minus the address of the destination buffer. This might overflow from one buffer into another, which is a problem, but it's less of a buffer.

There are many, many other proposals, e.g. StackShield, which copies return addresses into the data segment rather than the stack, and checks that the two are equal when the function is about to return. This can be implmented as an assembler file processor, but again it requires code to be recompiled.

Recently, Microsoft released a new way to handle this called control-flow integrity (CFI), which is a combination of static and dynamic checking; then, it can statically determine the control flow, and dynamically check that the address one returns to is within those options. This is reasonable, but doesn't work in all cases (e.g. if one returns to an address from a function pointer), and it's an active area of research.

We just talked about a lot of flags or compiler stuff; make sure all of them are turned on in your code, or it may be exploitable. This is a common lesson.

The Confinement Principle. It seems like there's lots of untrusted code out there that we need to run, e.g. apps or extensions from untrusted sites, exposed applications (e.g. PDF viewers and Outlook) vulnerable to attacks, legacy dæmons that are insecure (e.g. sendmail), honeypots, etc.

Often, we want to or need to run these, and want to guarantee that, even if they're compromised, only that app or program is compromised, and the rest of the system is unaffected. In fact, we could try to kill the application once it misbehaves.

Confinement can be implemented at many levels. The most basic way (the DoD or Pentagon approach) is called the "air gap:" make sure that an application is run on a completely isolated application or network. This is a very strong defense, though not foolproof: the user could plug a USB stick into an infected side and move it to the isolated side (or a laptop, etc.). The defense is also hardware-based: you use a jar of epoxy to seal the USB ports. Well then.

The next lowest level is to use virtual machines: isolate operating systems on a single machine. Then, you can check your mail on one VM and launch the missiles on another VM, so it's (in theory) all good. This is courtesy of a virtual machine monitor (VMM), though it adds some overhead, and sometimes they have bugs that allow malware to bleed across. However, it is a lot harder for malware to sneak across.

The next level is process-level confinement (e.g. browsers), where processes are less able to spread stuff between each other. Finally, one can implement it on the thread level (software fault isolation, or SFI), which is more interesting to implement because threads can share memory. This is also done in browsers.

All levels of confinement have a few commonalities. There is a *reference monitor* which mediates requests from applications and determines which things are allowed and disallowed, e.g. a certain application may be disallowed from touching `/etc/passwd`. However, it must always be invoked, and it better be tamper-proof (or the attacker can exploit it), so there's a lot of work around building simple, provably secure reference monitors.

The `chroot` program is an old and useful way to do this. It means that the given user cannot access files above the new root directory. An application called JailKit (or FreeBSD jail) makes this work; early versions had bugs, but FreeBSD jail is the right way to do it.

For example, this is really nice for webservers; they can be jailed to a certain directory, so if the server is compromised, it can't get anything other than that directory itself. It's good to be aware of the jail mechanism.

However, jails don't work so well for web browsers or email (where we want to upload or download files from arbitrary places in the system). The key is that jails are all-or-nothing; we would instead want to determine which kinds of access are allowed or disallowed.

System Call Interposition. System call interposition is confinement at the level of a process; thus, the only way it can damage a system or even change it is through the filesystem, i.e. through system calls (networks, writing, reading, etc.). Thus, we'll monitor all system calls and block the ones we don't like.

This can be implemented in many ways: in the kernel (e.g. GSWTK), in user space (program shepherding), or a hybrid of both. We'll look at a hybrid, Systrace.

Linux has a function called `ptrace`, which listens on a process ID and wakes up when that PID makes a system call. Then, it can signal the monitor, which determines whether it's OK.

There are several issues (nuances, complications?) with this.

- If the process forks, so too must the monitor, which is a growth in complexity.
- If the monitor crashes, the app has to be killed.
- The system calls depend heavily on context; for example, `open("passwd","r")` depends heavily on which directory we're in, so we have to keep track of state somehow.

`ptrace` isn't well suited for this application, since it traces all system calls or none. The monitor doesn't need to be woken up for `close`, for example. Additionally, the monitor can't abort the sys-call without killing the app.

Moreover, process monitors have a big security problem: race conditions. Suppose process 1 tries to open file `"me"`, and meanwhile, process 2 symlinks it to `"/etc/passwd"`. Then, both will pass the monitor, but it is a vulnerability. This is called a *TOCTOU* (time-of-check, time-of-use) bug, and it's less difficult than it looks to implement. Thus, it's extremely common in multithreaded environments. The issue is that the check isn't atomic to the execution; when implementing things, be sure to keep it atomic!

Systrace is a better solution; it lives in the kernel, and only forwards monitored calls to the monitor, which is more efficient, and it follows symlinks and replaces the paths, so the TOCTOU vulnerability is less possible. Furthermore, when an app calls `execve`, a new policy file is loaded, to deal with the staggering variety of corner cases.

A sample policy file:

```
path allow /tmp/*
path deny /etc/passwd
network deny all
```

This is nice, but is it used widely? No, because it requires the user to specify policies for all of their apps. In general, asking the user what to do is one of the worst things you can do; the user is not an expert, so it doesn't help anything. Security prompts are a very difficult problem — it's become a whole science, and we'll talk more about it later on. Systrace tries to learn good policies, but of course that isn't perfect.

NaCl (Native Client) is a modern-day example built into Chrome. It looks useful, but people don't use it much, I guess. This allows a website to send `x86` code to the browser, which directly executes it on the processor. This is a lot faster than the Javascript interpreter, which is nice, but of course it requires some security to deal with. . .

There is an outer sandbox which restricts the code's capabilities using SCI, and an inner sandbox which prevents it from accessing memory outside of its own alloted stuff.

There's also isolation via VMs, which we'll talk about next time.

Lecture 5.

Program Analysis for Computer Security: 4/9/15

> *"Would you* really *want to spend the rest of your life on a tropical island with someone bringing you drinks?"*

In case you weren't aware, software bugs are serious business; witness the Android phone owner who noticed that his phone crashed whenever he tried to call 911![6] This is both a very important bug to fix and one that's hard to easily test ("oh hello 911 again, no, we're just still testing. Sorry!") Another important bug example was when one guy found a simple way to delete a Facebook photo album using a vulnerability, and got a nice payout for discovering it.

An analogous example of a vulnerability isn't a software bug *per se*, but arises from people doing something wrong, e.g. the kind of thing that you could sell to *The New York Times* for a nice scoop. Should you tell the company before alerting the press? It depends. Is it an honest mistake? What if we're wrong and the company wasn't doing anything bad?

Bugs are important (embarrassing or vulnerable) for the developer, and frustrating or vulnerable for the user. So this is a nice motivation for fixing bugs. So, how do you know whether software is OK to run? Here are two options.

- Static analysis takes a piece of code and some information about what it should(n't) be doing, and then issues a report about some things that look weird, e.g. memory leaks, buffer overflows, SQL injection vulnerabilities, etc.
- Dynamic analysis (e.g. testing software) takes a piece of code and tries some inputs and sees what happens.

[6] http://www.android.net/forum/htc-evo-4g/68261-calling-911-crashes-my-htc-evo-4g-every-time.html.

Static analysis has some advantages, e.g. it can consider all possible inputs, and can prove the absence of bugs (things which dynamic analysis generally cannot do). However, dynamic analysis can make it easier to choose a sample test input (even guided by static analysis).

Of course, it's much less expensive to find and fix bugs sooner (development or QA) than later (maintenance).

Dynamic analysis can take two flavors, instumenting code for testing, e.g. Purify for heap memory, Perl tainting (information flow, in which certain data is considered "poisoned" and should be kept away from other pieces of data), and Java has race condition checking, which tries to determine whether an erroneous state is due to a code flaw or a timing error. But there's also black-box checking which doesn't require knowing anything about the code, e.g. fuzz testing or penetration testing, or black-box web application security analysis.

Static analysis is something with a very long research history, because it ties many disciplines of software development, and it's directly relevant to the compiler's analysis. It's also a nice thing to have, an algorithm that says "this code is correct," or at least something that's somewhere along the way, algorithms that help us write better code. And there's been a decade of commercial or open-source products of static analysis, e.g. Fortify, Coverity, FindBugs, and some of Microsoft's tools.

Let's talk about static analysis for a bit. First, what are the goals and limitations of these tools? We'll look at one example; there are many paradigms, but this one, Coverity, will illustrate a simple and elegant example, which mirrors code execution.

Two particular goals of static analysis are to find bugs, identifying code that the programmer wishes to modify or improve, and to verify correctness of code, e.g. that there are no buffer overflows, etc.

Definition.

- A static analyzer is *sound* if it finds all bugs, or, equivalently (since $A \implies B$ is equivalent to $(\neg B) \implies (\neg A)$), if it reports that there are no bugs, then there are really no bugs.
- A static analyzer is *complete* if every bug it finds is really a bug, i.e. if there are no bugs, the analysis says there are no bugs.

These are pretty nifty notions; we'd love for our analyzers to be both sound and complete. But maybe it's, like, nicer philosophically to have things to work towards, and for programming to still be interesting. Well, it happens that soundness and completeness is undecidable.

However, picking soundness or completeness (or neither) is perfectly decidable, e.g. reporting everything or nothing. So the theory isn't so hot, but the point is we can make reasonable approximations to soundness or completeness, and this is what we in practice aim for.

A program that is sound, but not complete, might report too many bugs. Imagine a piece of software broken up into modules. Because of soundness, we need to approximate their behaviors, but we need to look at all of them. In other words, understanding this program must take into account all of its real behaviors. Thus, it over-approximates the behaviors.

In some sense, we have a big box of behaviors, which is approximated with a bigger box. Then, the errors in the bigger box might be in the program, or only in the approximation. You can also see how complete checkers that are unsound work: they approximate the behaviors by underestimating them, and thus all their errors are real, but they might miss some outside their scope.

This execution-based approach takes code and visualizes it as states, with arrows between them representing execution paths. This gets complicated because of loops (man, why do you guys have to write code with loops in it!?) and because multiple arrows can converge on one state.

Then, at each block i of code, there's a *transfer function* d_i which takes in the state and returns the new state; for example, the code `x <- x+1` takes in a state with `x` and returns by altering it. Then, two blocks in turn is just function composition. The symbol for what happens when two paths meet at the same block is $d_i \sqcup d_j$, called "meet," and "join" comes up when a path can split.

A static analyzer can kep track of how much information it knows about a variable; then, $\top$ represents knowing nothing, and $x = \perp$ means we've found a contradiction (too much information). These can be refined into the *Boolean formula lattice*, where we have a lattice of `true`, `false`, and $y = 0$ and $y \neq 0$; then, `false` is the contradiction and `true` is the state where we know nothing, and there's a *refined signs lattice* that tracks whether $x = 0$, $x > 0$, $x < 0$, $x \geq 0$, or $x \leq 0$; the lattice structure is given by `or`, i.e. $x \geq 0$ has arrows from $x > 0$ and $x = 0$.

Now, using these lattices, one can trace through a program and assign what can be known to a program at every point, and use this to fil in more information elsewhere in the program, like a crossword. Thus, one (e.g. a static analyzer) can use these tools to prove things about a program, e.g. that it doesn't crash.

An unsound analyzer, however, can restrict its scope to the kinds of errors that one thinks developers are most likely to care about. For example, the `chroot()` protocol checker intends to use `chroot()` to ensure that a program only has access to certain directories, but if you don't change the current working directory, then, oops. Thus, a protocol checker could go through to make sure that step is taken too, and if not issue an error. Clearly, this isn't sound, but whenever it reports a bug, you'd better go fix it.

Another category of checkers is called tainting checkers, which check where untrusted data is used. The key is to analyze a function before analyzing its caller, and to be practical, it's good to have summaries of standard library functions, to save time. Thus, one can detect where the memory is used.

Given some source code, one can make a *control-flow graph* (CFG), and then run individual checkers (e.g. for buffer overruns, null pointers, and use after free) on the graph. However, of course there would be false positives, e.g. false paths. The control-flow graph might contain paths that will never be executed, e.g. if one knows that an integer might live within a given range, but can't convey this to the static analyzer. Then again, now that we know this, we can implement range checkers.

Another good principle is that "most code is mostly right;" for the most part, everything in the code has a bug in it, but the code basically doesn't fail that badly, for the most part. This is a very fuzzy heuristic, but when well done can reduce the tens of thousands of error messages to something more useful.

When these tools were run on Linux and BSD, people found noticeably many more bugs in Linux than BSD, even serious security vulnerabilities! It turns out this was because there were more device drivers, and both of them are about as secure.

One recent application of this, e.g. by the professor (Mitchell) is program analysis, specifically tainting analysis, for Android apps. This would be hard for source code analysis, because the source code relies so heavily on other libraries, etc. Thus, these libraries were considered as models, using a combination of static and dynamic analysis, in order to make this tractable.

A particularly useful application of this is to determine what exactly goes on in an app's permissions; for example, it may request Internet access to sync Facebook contacts, so it should only access the Facebook API. But what if it also sends your contacts somewhere on the Internet? (This really happened in one case.)

Lecture 6.

Secure Architecture: 4/14/15

"I learned something about timing and students. My son is in college on the East Coast, and I found that our schedules are the same."

Isolation and Least Privilege. This is the notion of putting things in boxes and separating them, so an attack on one component doesn't lead to attacks on others. Another useful concept is defense in depth: if you and Bob are arguing as to which security principle is better, why not do both? That's better yet. Finally, keep in mind simplicity and to fail securely (e.g. a crash is better than root access, etc.).

The *principle of least privilege*, which seems a little obvious in retrospect, is that every component of a system should be given only as much privilege as it needs to do its work. This leads to an idea that the same stuff shouldn't be in control of the network and the filesystem, because then a network vulnerability leads the adversary to be able to read the filesystem.

Thus, it's better to keep everything modular, in separate components that communicate, so if an adversary Mark Yuckerberg wanders into the networking module, it can't do anything more than the API calls that the networking program already had, which is not ideal but better than unfettered access. In this context, a privilege is an ability to look at or modify a resource in the system.

For example, the Unix `sendmail` program has a very monolithic design, and is historically a source of many vulnerabilities, since it doesn't separate its networking parts from its filesystem parts (inbox, etc.), and it has a big list of buffer overflows. An improved design called QMail uses the principle of least privilege.

Recall that in an OS, processes are isolated: each process has a UID, and two processes with the same UID have the same permissions. Thus, a process may access files, sockets, etc., depending on what permissions were granted according to its UID. This is an example of compartmentalization: each UID is a compartment, and defines privileges. In QMail, each module is run as a separate "user." The principle of least privilege encourages minimal privileges for each UID: there's only one program that can set UIDs, and only one root program (i.e., with all privileges).

QMail has eight processes that communicate in a tree structure: two for reading incoming external and internal mail, one for the queue, and five components associated with sending mail (a master, and two each for internal and external mail). Each runs under a separate UID; for example, the queue utility, `qmailq`, can read and write the mail queue, but not all that much else. It reads incoming mail directories and splits the message into a header and a body, and then alerts the `qmail-send` utility, which determines which of its local or external processes to run for an outgoing message. In particular, as few as possible are given root privileges (or any other unnecessary privileges, for that matter).

Another more recent example is the Android application sandbox. Each app not only runs in its own UID, but also in its own VM! This provides memory protection, and communication between apps is limited to Unix domain sockets. Moreover, none of them, except for `ping` and `zygote` (which spawns new processes) can run as root. Then, interactions between apps requires a reference monitor to check permissions. Since apps announce their privileges at install time, the principle of least privilege once again applies. See Figure 1 for a schematic.

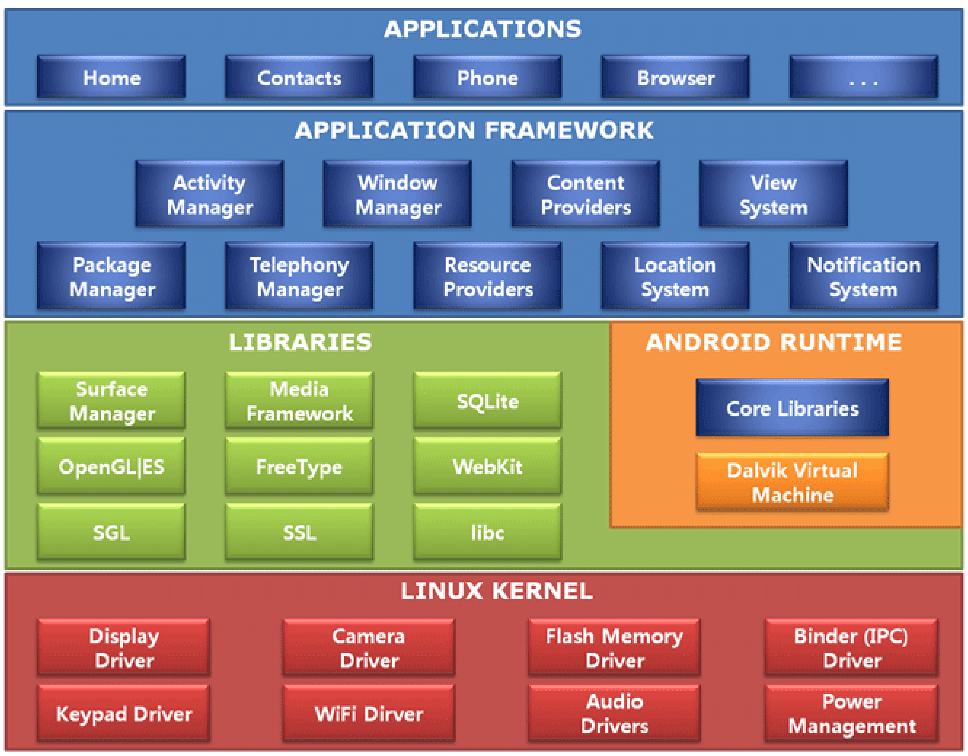

FIGURE 1. A depiction of the components of the Android OS. Different apps not only use different instances of the application framework, but also of the runtime. Source: http://www.cubrid.org/blog/dev-platform/android-at-a-glance.

Access Control. In access control, the idea is that some sort of reference monitor governs all user processes' access to resources. This has some big assumptions: it requires isolated components with limited interaction, so that there's no way for processes to access resources without going through the reference monitor (which has sometimes been the tragic flaw in some implementations of access control). Moreover, we need to know who the user is (via passwords or otherwise).

Now, when you get into a field early, you can have totally obvious things named after you, like Lampson's access control matrix. Each subject (user n) is a row, and each object (file n) is a column, and the intersection of row m and column n is the kind of access that user m has to file n (e.g., reading, writing, both). In some sense, we have subjects, objects, and verbs (the privileges). There are lots and lots of papers on access control, because the Department of Defense took an interest in it.

There are two main ways to store these: by column (access column list, or ACL), i.e. per file, or by row (by process). In an ACL, each user has a "ticket" for each resource, and they have to be secured somehow (e.g. using cryptographic protocols). And, intriguingly, since the reference monitor checks only the ticket, it doesn't need to know the identity of the process.

Typically, when a process creates a subprocess, it gets the same UID. However, it's not that easy: we'd like to be able to restrict the privileges of subprocesses; this is a reason that we prefer capability-based systems. This also makes it easy to pass privileges between systems (as parameters, or over a network, or whatever), whereas adding to an access control list is hard. On the other hand, revoking an access is easy if it's file-based (ACL), since if I own the file I can just remove them from the list. But finding people who have a ticket is hard, unless a ticket is a pointer to a switch I can turn on or off.

Finally, one can use group IDs rather than user IDs (e.g. role in an organization), which can be arranged in a hierarchy (i.e. poset), which make privilege-setting relatively easier. In particular, it's much easier to add or remove users. This is called role-based access control. There's a guy running around who thinks he invented this, at a different university, but apparently it's pretty obvious, so whatever.

Operating Systems. In Unix, access control is through processes' user IDs. Each process can change its ID, with a restricted set of options. Then, the root has all permissions. Each file has an owner and a group, and the owner can set its permissions (read, write, and execute for the owner, for the group, and for other, stored as a sequence of bits) — the root may also change permissions, but that's it. There is a utility for setting UIDs, called `setuid`, but unless you're root this has an extremely restricted set of options. These steps, for example, prevent an adversary getting into many systems from one.

One drawback is that it's too tempting to use root privileges, since the permission system is relatively simple. Thus, programs that run as root don't always need to, or could be more compartmentalized, or so on, and this makes buffer overflows more fruitful exploits.

Windows functions similarly to Linux; there's users, groups, and UIDs, but it's a little more flexible: there are also tokens and security attributes, and the result is that it's a lot easier to downgrade one's one own privileges (no need to do gymnastics like the dæmonized double fork in Unix). The token is akin to the tickets described earlier.

Instead of a user ID, it's called a security ID (SID), but they function similarly. A process executing has a set of tokens, called its *security context*. One could be the SID, and there could be impersonation tokens that allow it to run as different users to temporarily adopt a different security context. These tokens are inputs into the security monitor.

For example, an access token might say that this user is a member of two groups (e.g. admins and writers). Then, if it tries to write a file with a given security descriptor, the descriptor contains an entry in the discretionary access control list (DACL) that can explicitly deny or allow certain users or groups.

Explicit denies take priority, then explicit allows (we're trying to be conservative), followed by inherited denies or allows. (Inherited permissions come from hierarchies among groups.)

Impersonation tokens can be created by the process that is being impersonated, and then it can pass them to the process that should do the impersonating. These can be completely anonymous, for identification, more complete impersonation, or delegation (impersonation on remote systems). The point is, this is nuanced, which is good, but also complicated.

This isn't all sunshine and roses, e.g. registry-based attacks.

Browser Isolation. A browser is like another operating system; it executes code on pages or frames. Thus, let's implement security on browsers.

Different frames can interact. Can Frame A execute a script that manipulate arbitrary elements of Frame B? Can it change the origin (navigate) the content for Frame B? The browser's "kernel" is given full access, but other things, e.g. plugins, run with reduced permissions.

This is able to leverage the OS's isolation principles; for example, the rendering engine does some intelligent downgrading of its privileges as it runs. This was implemented in IE first, and so for a while, IE was a bit more secure against arbitrary code execution vulnerabilities (CVEs), especially in the rendering engine, and now this has spread to other browser designers.

Lecture 7.

Security Bugs in the Real World: 4/16/15

"Star Wars is a series of movies that used to be good. But you wouldn't know that."

Today's guest lecturer is Alex Stamos, the chief security officer at Yahoo! Before working there, he was at several other companies (not at once, I think), including one he founded, and before that, he went to Berkeley. Today, he'll talk about the economics of security bugs, with real examples; bug-finding techniques; bug-finding as warfare; and how to leverage this into a lucrative career.

2014 was a huge year for bugs, in things as fundamental as OpenSSL, `bash` (a bug that was dormant for 17 years, which is amazing!), Apple and Microsoft products, and more. These bugs are fundamental and widespread, which is a huge headache: they have to be patched on a huge number of systems quickly, and without taking a company's product offline. Of course, quick fixes of these bugs cause other things to break, which is, uh, interesting.

One lesson from this is that the big layer cake of abstractions has only been looked at from the very top and the very bottom, but offensive and defensive actors have started looking into the middle.

There have also been lots of breaches, e.g. eBay (which is scary because lots of people use their eBay usernames and passwords on lots of other accounts), P.F. Chang's, and, most surreally, the attack on Sony's *The Interview*. This was (according to Stamos, according to a briefing he couldn't be more specific on) an attack acting under the North Korean government. One lesson here is that North Korea isn't a huge cybersecutity player, and can't mount attacks on the order of China or the United States; however, private companies, even big ones like Sony, aren't really prepared to deal with attacks on the second- or third-tier level.

There were also lots of attacks that were heavy in user impact, e.g. the iCloud leak, which allowed some people to leak private pictures of celebrities; the private impact of stealing content from others' iPhones was likely much greater, and we actually don't know. Apple fixed the bug, of course, and tried to cover up the whole thing. These kinds of things are the scary ones: credit card theft is a headache to a bank, but less so to the cardholder, but this kind of vulnerability, leaking people's photos or otherwise private information, is much more impactful on the personal level: a malicious actor could ruin someone's life in many ways. This is especially important at Yahoo!, since they have the email client.

So who finds security bugs? Engineers like to find security bugs: the idea is that software is probably going to be flawed, so they want to cover as many flaws as possible, to reduce the instance of exploitation. Coverage metrics (which are tricky to pin down accurately and completely) are a reasonable way to find some of them. Engineers have access to the source, debugging environments, and so on, and having written the code, they understand it well, and are a good way to find bugs, but conversely, they are more blind to their own mistakes relative to outside actors.

However, people prefer to care about features rather than security, at least before things go wrong. And engineers are often not judged on the security of their code. This, along with the collaborative nature of writing code and the incredible complexity of modern systems, means that it's basically impossible for any one person to understand enough code to comprehensively call code secure. Even Adobe Acrobat has probably 20 *million* lines of code. To just display a PDF! This code bloat is partly due to some big features for big customers that not many other people use, and these can introduce security vulnerabilities. For example, Acrobat can embed Flash applets into PDFs (and therefore open one PDF inside another PDF), has a whole Javascript interpreter, and even contains a Web browser![7] Most worryingly, there are video codecs, which are dangerous because they run heavily optimized C or C++, and therefore are almost impossible to make secure.

Well, this seems pretty hopeless. So the solution is basically a jail, where Acrobat exploits can't do too much to other parts of the computer.

Criminals also look for bugs. They want money, and to stay out of jail (i.e. they want reliable bugs that don't expose themselves). Another issue for these actors is that they have a finite amount of time and money, and all decisions need to have a good profit margin.

[7] Wow, this is almost approaching the feature bloat of Emacs.

Security researchers also look for acts: they want publicity/recognition, preferably in a trendy area. They have good access to engineers, but there are fewer and fewer of them nowadays; they tend to be private now, selling exploits to companies.

Pen testers look for security bugs without knowing the code. Like engineers, they want to get good coverage, though they don't need to worry about fixing the bugs, just finding them. They have access to source code and engineers, as well as permission, which is good.

There are governments, which we'll talk about later.

We also have hacktivism, which has gotten less popular as people have gone to jail for DDoS attacks. They want to do something "good," while also staying out of jail, so they've moved to more targeted and less criminal attacks. They might not be as knowledgeable on the code or techniques as pen testers or engineers.

Academics are interested in security, too, but they theorize, trying to find common flaws and broad classes of techniques, or to develop new crypto. But they have a longer-term time scale for this.

Bug Finding. There are two kinds of testing: black-box (no source) and white-box (you can look at the source).

In black-box testing, you probe the program and look for errors, and then try to exploit them. One example is fuzzing, which is basically a walk through the state table for a piece of software; since software is so complicated nowadays, there are way too many states to really test this rigorously, so one might hope to find a path that nobody else has found before.

One way to think about this is as a series of concentric circles: the innermost is the state space that the engineer things can happen (what they had in mind), the next is what the QA tester finds (since they test things the engineer might not expect); outside of that is what the security researcher looks for, which tries to find things that nobody expected, e.g. unreasonable input. For example, what if we tell the form that our user's surname is `O'DROP TABLES`?

It used to be that you could do fuzzing somewhat carelessly, by feeding random data to places, but now you have to be a little smarter, because people have gotten a bit better about, e.g., sanitizing input to Web forms. For example, randomly flipping bits in `.docx` files is kind of unhelpful, because they're just zipped directories, so we'd only be testing the decompressor. So we want to check specific things and compress them, and for each format (e.g. video codecs) there's good and bad ways to fuzz them.

People tend to keep the best fuzzers to themselves, to make money off of their hard work, but APLFuzz and PeachFuzz are good, public fuzzing tools.

Debuggers are also useful for checking whether security bugs are important; some of them are caught before they can do anything dangerous. Related to this is the idea of reverse engineering, e.g. decompilation of high-level languages (.NET, Java, etc.), and sometimes C or C++. Disassembly is also pretty nice, and for people who are experienced, this is probably one of the best ways to do this. For example, you can search for known dangerous sequences of instructions, and then build a call graph of paths that go to that code and whether it's possible to exploit them.

One interesting way to find bugs is to reverse patches, e.g. using a tool called BinDiff, which compares the assembly code of the binary before and after the patch. This means people could download the patch and try to figure out its impact on other parts of the program before the company does. Stuff like this is why Microsoft waits a few days before getting out its patches.

It's possible to restrain or defeat black-box analysis, e.g. programs that determine whether they're being debugged or virtualized (e.g. see how long interrupts take. Were they caught by debuggers?), and then respond accordingly. Alternatively, a program can ask the OS whether the debugger is in operation on it. There are also methods to make code harder to reverse-engineer code, e.g. packing code in shared libraries, doing pointer arithmetic, and encrypted or obfuscated code. This has its own issues, e.g. providing lots of fun security leaks, breaking lots of modern memory protections, and confusing malware detection programs which look for heavily obfuscated code.

There are also anti-anti-debuggers, e.g. the Windows DRM manager connects with Windows the first time it's run to relink everything into one's own personal binary with a unique global ID. This ends up meaning that an exploit on one computer is fine, but doesn't work on any other copy of the program. These kinds of things tend to slow attackers down, rather than stopping them.

Black-box techniques always work better with targeting, which leads to white-box methods (gray-box methods?). These are more helpful, e.g. better understanding of the code or better targeted attacks. But it finds a lot more bugs, which makes it harder to find signal from noise (e.g. an SSL bug on iPhones that nobody caught for years!).

So here's an example of a bug in the wild: let's look at session cookies. HTTP is a stateless protocol, so information such as your username and password, which is sent via an HTTP POST request, needs to be kept as state. This is kept in a "cookie," a little piece of data that represents encrypted security information. The cookie is different for different users, of course. This becomes a crypto problem, where the server has to decrypt the cookie and the user can't, so it can determine the user's name and time of the cookie (just like real cookies, these ones can expire too). So errors in the crypto cause security flaws, too, e.g. just xoring two usernames to undermine RC4 encryption.

The moral of the story is that you never use encryption for integrity protection: encryption $\neq$ protection. There's a wide world of security protocols, and there are better ones for this sort of stuff. Furthermore, they didn't understand how RC4 works, which, as you can imagine, a problem. A small website can respond by salting each cookie with a random number, but there will be collisions between numbers on large websites fielding millions of connections at once. Thus, they have to use signatures (like Yahoo! does, with ECC) or sometimes other techniques. Encrypting can also be done, to prevent people from mucking about with their cookies.

There are also OS-level bugs. Look at the following code.

```c
void attachSuffix(char *userinputprefix, size_t inputSize)
{
    char *withSuffix;

    withSuffix = malloc(inputSize + 2);
    if (withSuffix == NULL)
    {
        // meh, don't care that much
        return;
    }
    memcpy(withSuffix, userinputprefix, inputSize);
    withSuffix[inputSize] = '.';
    withSuffix[inputSize + 1]] = '\n';
    // ...
}
```

Here, `size_t` is an unsigned type whose type is large enough to store any address on the system, which is useful because the same code can be written on 32-bit and 64-bit systems. But that means large numbers can be overflowed into small numbers, causing it to return, say, a single byte of memory, and therefore overflow the heap, copying data into places it shouldn't.

Another curious example is a grammar checker which checks whether there are two spaces after each period. One forum used a very bad $O(n^2)$ algorithm, and had a character limit of $10,000$ characters. Thus, submitting $10,000$ periods made it hang for a long time...

One interesting consequence on the defensive side is that innovations in warfare tend to decrease the cost for later adopters. For example, the Germans spent lots of time and many engineers in World War II making an assault rifle called the STG-44. But then a soldier named Kalishnikov picked one off of the battlefield and was able to improve on it without nearly as much effort, creating the AK-47. It's much simpler to build, even in a garage or with a 3-D printer. The point is, the first time, it was really hard, and lots of engineering problems had to be solved; now, it's much easier.

The nuclear bomb is another example; it took the best American physicists and engineers many years to develop the math and make the first bomb, and then it was taken by the Soviets and the PRC. Fortunately (ahem), this requires a huge industrial base to make a bomb, so it hasn't trickled down that far (yet).

In cyberwarfare, things are little different. APT attacks, which start with personalized (spear) phishing and then using various techniques to spread the attack horizontally on a network, were on the level of a nation-state in the mid-2000s, but now they can be carried out by, say, any Eastern European gangster in a tracksuit.[8] The idea is that no industry is needed for cyberattacks, so once it's been done once and shared, it can be spread and reimplemented really easily.

[8]Whenever the *Wall Street Journal* talks about Chinese hacking, they run a stock photo of a bunch of uniformed Chinese soldiers at computers; a closer look at the photo reveals it's a PLA Starcraft tournament.

Here's some more code, from the Aurora attack code.

```
function window :: onload()
{
    var SourceElement = document.createElement ("div");
    document.body.appendChild (sourceElement);
    var SavedEvent = null;
    SourceElement.onclick = function () {
        SavedEvent = document.createEventObject (event);
        document.body.removeChild (event.srcElement);
    }
    SourceElement.fireEvent ("onclick"); // creates an event handler
    SourceElement = SavedEvent.srcElement;
}
```

This bug relates to how things are stored on the heap, and before IE had process isolation, allowed people to gain complete control over IE. This led to "Operation Aurora," which used social engineering (the weakest link, always) to get someone to click a link, and then escalated privileges using this. Then, it spread, and allowed the attacker to read the important data, which was emails and financial data. This was much easier than getting the data directly, since Windows is the same everywhere, and depends on only one person messing up, rather than confronting the relatively secure email server. This is a typical nation-state attack, in that it goes through a corporate network: the research can be amortized, in the sense that once it's written and implemented, it can spread for a bit, and then hit many companies at once (Aurora hit 35 companies at once!).

Lest you think the Chinese are the only ones to do this, recall Stuxnet, which was likely developed by the United States and Israel and singlehandedly set back the Iranian nuclear program by several years. This might have averted a war, which is kind of insane.

Stuxnet contained *five* zero-days, which is insane: that's a very, very expensive bug. It included two rootkits and a few worms. Even though the nuclear computer networks were airgapped, someone plugged a USB stick into the computer and their own computer, and, well, oops. It was spread all over Iran, just for the purpose of getting to the plant, and had specialized software to check whether the code was running the nuclear centrifuges; then, it ran code to physically damage the centrifuges (and lie to the control system to make this harder to detect), which is why it took so much time to recover. Impressively, Stuxnet also used the same way to get information out: it created files that spread back out into the global network, and could therefore report back to the President. These are scary impressive: another worm embedded itself in hardware!

So the question is: if you find a bug, will you disclose it responsibly (to the company) or fully? Everyone in this class has the privilege to choose a job they agree with, to a degree, and therefore you can't say that you're just trying to feed your family, since it is probably possible to get another job that isn't as amoral. Relatedly, don't go into a big company blindly and overspecialize, so that you can't go somewhere else. This is a good reason to work at a smaller company (running servers, DevOps, and so forth). But on the other hand, most startups fail, so don't naïvely start your own company: you don't know what you don't know. So if you're doing security as a career, think about what you want to do, what your morals are, and so on.

More advice:

- Before every meeting, spend 30 seconds to know what you want to get out of the meeting. This will help you to get more out of the meeting. If you don't know why you are having the meeting, cancel it!
- If you get an offer, try to negotiate. Ask for a couple days to get back, and then, on the callback, add maybe 5% more (or more, when you're better). This is standard; they'll negotiate too, and sometimes you'll get the raise. Almost always, you'll still have the offer (whether at the original price or a little more), and if a small company can't make a cashflow, ask for more equity.
- Common stock is apparently for commoners. This is last in the line for making equity decisions, so if you work at a startup, ask for founder shares, since this means the same decisions (and same profits or failures) as the founders and the investors.
- Every company is a tech company right now, but you want to be writing products, not plumbing, especially if you specialize. You want your specialty to be the focus of the company: otherwise, nobody will think about you until it breaks, and you'll never be in a position to change the company.

Lecture 8.

The Browser Security Model: 4/21/15

"I think the interesting thing about this field is that there's a lot of paranoid people."

A good lesson from the first project is that you should never say, "that's too tricky, nobody would ever do that in the real world," because people will and do.

We'll spend a little time talking about web attacks; they're a large and popular class of attacks, e.g. cross-side scripting vulnerabilities. However, since about 2009 there's been progress, and now there are fewer such attacks, and more system-based attacks. There's an ever-fluctuating mix of different kinds of vulnerabilities, and we'll learn about each in turn. And even though the proportion of web attacks is going down, that's because the amount of other attacks is increasing (especially mobile ones). This is the first of five lectures on web security in various forms.

Web security has two reasonable goals.

- *Users should be able to browse the web safely.* They should be able to visit without having information stolen, or one website compromising a session of another site.
- *Servers should support secure web applications.* This means that applications deliviered over the web should be able to achieve the same level of security as stand-alone applications.

Notice this doesn't include protecting the user from bad sites; that's more or less their own problem, and much harder to protect against.

The *web security threat model* explains that there's a property we'd like to preserve (e.g. the balance of my bank account), and an attacker has limited capabilities: it can set up bad websites that the users end up visiting, but it has no control over the network. This adversary is called the *Web attacker*.

The *network threat model* allows the dastardly villain to control (some of) the network rather than the end sites, and can read or affect both directions of traffic. This is a much stronger attack. In this class, we'll discuss security models that are safe against one or both models, so these terms are good to have in mind. These attackers are further divided into passive and active (whether they can read and write, or just read).

The malware attacker can escape browser isolation mechanisms to run independently of it, and directly under the control of the OS.

We all know what a URL looks like, e.g. `http://stanford.edu:81/class?name=cs155?homework`. The different parts have standardized names, e.g. the protocol, the hostname, and several more.

HTTP requests are of the form `GET` and `POST`; the former should not have side effects, but the latter may. HTTP requests can also have referral information (e.g. in the case of page redirects) — in this case, it's often quite useful to know that the referrer is what you thought it was, e.g. login information or bank websites.

The browser's execution model divides a page into a bunch of frames: this is the unit of execution. Each frame or browser window is loaded and rendered (can involve more requests, loading images, subframes, and so on).

Here's some sample code.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<button onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

You can do a surprising amount even with just HTML, e.g. any of the applets at `http://phet.colorado.edu/en/simulations/category/html`. These generally use HTML5, and they even work with touchscreens.

The *document object model* (DOM) is an object-oriented interface used to write documents: a web page in HTML is structured data, so this is a reasonable approach, and the DOM provides an interface to that structure. For example, there are properties, such as `document.URL` and `document.forms[]`, and methods, such as `document.write(document.referrer)`. This is a superset of the browser object model (BOM).

Here's another example, using a very small amout of JavaScript.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<p id="demo"></p>

<!-- why 11? Go watch Spinal Tap -->
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

The web has had image tags since pretty much the very beginning, via the `<img>` tag. If you're a paranoid security person back then, there are a lot of issues there: the image code could have some buffer overflows, or so forth. Moreover, the browser tries to help the user, so things which aren't images get processed too.

But the most important point is that the request for the image is sent out to another site on the web, so there's a communication channel out from any web content, e.g. image. Thus, it's impossible to prevent that information from getting out, even if it's sensitive. Oops! And then it might as well have been stolen and broadcast to an adversary. In particular, any argument or return value of a Javascript function can be used to compute the URL of an image that a user requests.

The web wasn't invented by security people, and it's a great place to be an attacker.

Javascript also has error handing and timing, so it can request images from an internal IP addresses, e.g. `<img src="‘92.168.0.4:8080">`. Then, it can use timeout or `onError` to determine success or failure, and therefore know what's going on server-side, even if there's a firewall between the malicious server and the browser.

Another basic thing is remote scripting, with a goal of exchanging data between a client-side appication in the browser and some sort of server-side app, but without the user reloading the page. This is a bidirectional path of communication, which by now should strike you as suspicious. This can involve several models, e.g. Flash or a library called RSI. Typically, every (say) ten seconds, the browser asks the server what happens next. Thus, once the browser visits the site, the server can update that code in perpetuity.

We think of websites as isolated: the frame (or iFrame) is the basic unit of isolation. You can delegate part of the screen to content from another place; furthermore, if one frame is broken, the page around it may display properly, whoch was not. Another example is popped-out open chat windows.

So, while it's good for frames to be relatively isolated, it's also helpful to have them communicate, which makes for nice things like email/chat clients. This sounds like isolation of processes on the OS level!

The security model gives each frame an origin, the website it came from (or more specifically, part of the URL: protocol, hostname, and port number) — the same port is considered the same frame, more or less. Thus, they have access to each others' DOMs, but other frames from other origins do not.

There are some components to this policy, e.g. `canScript(A, B)`, the relation that `A` can execute a script that manipulates the DOM in `B`. There's also `canNavigate`, which specializes `canScript` to reolading, and there are protocols for reading and writing cookies. And if you import a library, that has to be considered, too.

The web is filled with these little quirks and gotchas. For example, should `www.facebook.com` and `chat.facebook.com` be considered the same frame? There's a Javascript command which allows a change of origin to just `facebook.com`, so the two can have the same origin. The basic idea has lots of details to it.

How do frames communicate with each other? One way is `window.postMessage`, an API supported by all standard browsers (IE, Safari, Firefox, Opera, Chrome). It is a network-line channel for frames, where one frame can send a message to another, which may choose to read it. This is a way to keep information going only to trusted frames. The protocol includes some information about the origin of the target, so that if an attack frame reloads its inner frame (which submitted the request), it isn't compromised; that is, the protocol requires the origin of the frame to stay the same throughout the whole transaction.

There are also attacks rooted in concerns about navigation. For example, there's the notion of a "Guninski attack," where a legitimate page is run in a malicious frame. The login page of the legitimate page is in its own page, so the malicious page can reload the frame with something else malicious, and thereby steal your login information.

So, when should one frame be able to reload another? You can say "never," but never say never: there are cool programming techniques where it's useful to have. Should frames be able to reload child or sibling frames? How about frame-busting, where it relaods the outermost frame? (This can be useful for promoting a small frame to the whole window.)

Different browsers once had different policies, with not to much sense to them. For example, reloading sibling frames leads to spread of malicious code, but the descendant policy makes sense. Now, though, browsers' policies are more consistent with this idea.

Another good question is, given a web site which looks like your bank's site, is it safe to enter your password there? Does the site have some crypto checked, with the bank's usual website? Or is it a different website? What if the URL changes by one character, and there's no certificate? Sometimes being cautious is also unreliable: the bank might contract out to a website that looks bad, but is authentic.

And of course, you have to be aware of a legitimate site running as a frame within a malicious site.

There are a lot of other mechanisms, e.g. messing with the status bar.

Cookies, which are data from the server stored in the client, can be used for authentication, where the cookie was saved so you don't have to log in each time. This is useful for user authentication and personalization, and some bad stuff like tracking, but we still want to protect them. For example, there are secure cookies: if you mark a cookie as secure, it will only be sent over HTTPS, which is encrypted. This is nice if you have sensitive information in that script. There are also HTTP Only cookies, which are important because they can't be accessed by Javascript, which helps prevent cookie theft via cross-side scripting (XSS). It doesn't do everything, but provides partial protection.

Frame busting is another technique: some sites want to ensure that their content isn't running in a frame around another site, i.e. so they're not victims of some other site's malicious frame reloading. The code for this is really simple, checking if we're not in the top frame, and then replacing the top frame if so. Despite the somewhat sketchy name, this is a legitimate practice, though it's hard to write great frame-busting code: sometimes it can be defeated by malicious outer frames (e.g. checking for frame-busting scripts and then killing them). It's a good thing to not do by hand; it's another constant attack/defend cycle.

Lecture 9.

Web Application Security: 4/23/15

According to a web security organization called OWASP, the most common web security vulnerabilities are SQL or SQL-like injections, which are relatively easy to fix; authentication session vulnerabilities; cross-side scripting (which is probably the most subtle and difficult kind); various implementation problems, such a misconfiguration or exposing sensitive files; and finally (in 8^{th} place), cross-site request forgery, which we'll also talk about today. This is the kind of thing that cookie theft falls under.

Code Injection Attacks. Command injection uses `eval` commands to execute arbitrary code on the server. For example, look at the following calculator in PHP.

```
$in = $_GET['exp'];
eval('$ans = ' .$in . ';');
```

This is pretty great, because you can execute whatever you want, even `rm *.*`.

These aren't usually as stupidly obvious.

```
$email = $_POST["email"]
$subject = $_POST["subject"]
system("mail $email -s $subject < /tmp/joinmynetwork")
```

This is bad because we're evaluating arbitrary system code. For example, one can call it with a URL that reads the password into the email subject.

These are most common in SQL, because many, many websites have a frontend and a database, and SQL is used as the glue to make things work. But it has to be sanitized, or one can inject malicious code.

The basic model is to submit malicious code, which causes the server to execute an unintended query, and therefore either cause damage or reveal stuff to the attacker. For example, CardSystems was put out of business in 2005 because of a powerful SQL injection attack in 2005 that stole millions of credit card numbers. More recently, there are many SQL vulnerabilities in Wordpress sites.

For example, is this code vulnerable?

```
set ok = execute("SELECT * FROM Users WHERE user=' " & form("user") & " ' AND pwd=' " & form
    ("pwd") & " '" );

if not ok.EOF
    login.fail
endif
```

So normally you send your username and password, which is as normal, but if you send as your username something like `or 1 == 1 ' --`, this always evaluates to true, and comments out the rest of the line; this forces it to always succeed, which is, as you can imagine, a slight problem. And you can do worse stuff, such as executing code within the statement that gains a root shell or deletes lots of stuff.

Relevant `xkcd`: https://xkcd.com/327/.[9]

So, how do you defend against SQL injections? It's extremely hard to make commands foolproof yourself, and most web frameworks have ways to do this correctly (even PHP!), to make sure everything is properly escaped. In this case, a command is a data structure, so it's a lot harder to mess up and allow someone to run arbitrary data. Alternatively, the command is parsed before the user and password strings are added, so that they're always treated as strings. Strongly typed languages also make this easier, because anomalous queries become type errors.

In other words, to prevent this category of attacks, just do modern programming with databases. But existing websites may still be unsafe. And many of them have huge, complicated logic around them to prevent SQL injections, which only makes it harder to tell whether they're secure.

In summary: SQL injections have always been at the top of the web vulnerability list by frequency, but they're oh so preventable.

Cross-Site Request Forgery. The most common case for this has to do with cookies and authentication.

Recall that cookies are used to authenticate: each time you make a GET or POST request, your cookie is sent along to ensure that it's really you. However, if the user visits another malicious server and receives a malicious page, that page could retrieve the cookie for the first website, since the cookies are sent for every communication. Therefore the malicious server has the cookie and can use it to impersonate the user to the original server.

You, dear reader — how long do you stay logged into your email, or to Facebook?

Here's a more specific example: suppose the user logs into `bank.com`, so the session cookie remains in the browser's state. Then, the user visits another site with specific HTML code which contains a form that's rendered on the victim's browser, which causes the user's browser to send that form, with that cookie, to the original site (for example, the form could send code to send $100 to the attacker).

Notice that the cookie isn't stolen by Eve, the malicious server; it just sends the form, and the user haplessly supplies its own cookies. This stretches the baking metaphor, though I guess that's the way the cookie crumbles.

In fact, you don't even need cookies, e.g. one could use this to attack home routers: a computer sets up its home router, and the malicious website uses this ability to reconfigure the router. Thus, when you're buying routers, try not to buy the one where anyone on the internal network can change the configuration! But simple consumer products have some default password, like `password`, so sometimes attackers just use this password, and therefore you should change the password so as not to be vulnerable. This is somewhat surrealistically known as a *drive-by Pharming attack*, and is an illustration of how CSRF attacks are a broad class of attacks, still popular in 2015.

So, how do you defeat it? You have to be sophisticated in your web programming and session management. The simplest way to do this is to add something more than just a cookie or a password to identify a password;

[9]Ironically enough, this site's HTTPS isn't working this week.

one is a secret token that can be placed within a page. This is a hard-to-guess or hidden part of the page, and authentication requires the session to respond with the same token. Many sites do this, e.g. the Rails home page.

These are simple in theory and hard to program, but many web frameworks, such as Ruby on Rails, have methods to handle this automatically.

Facebook and some other places use referrer validation, checking whether the referrer header for a form came from `facebook.com`; otherwise, it rejects the login. This works well, though not all forms come in with a referrer header. One can be decide to be lenient or strict, so the story can be more complicated. Some companies block referrer headers from outgoing traffic, which makes strict checking not so helpful. However, referrer heading suppression over HTTPS is actually quite low, according to a study by Professor Mitchell et al., so perhaps it's not a big deal after all. (Companies usually block their referrer headers because they leak information about what the company is searching for or doing on the Internet.)

Here's yet another CSRF type (which actually happened at Stanford): the attacker uses the victim's credentials to log in as the attacker. This means that the attacker can do what it wants (even searching for llamas or other terrible things, in the slide's example) and pin the blame on the user, and if the user has privileges, then so does the attacker. This is an example of an attack which isn't possible just with malicious websites.

This led to the design of another header called the Origin header, which is an alternative to the referrer, but has fewer privacy problems and supersedes it while defending against redirect-based attacks.

In summary, defeat login-based CSRF with referrer/origin header validation, and have login forms submitted over HTTPS. For other things, it would be useful to use Ruby on Rails or another method for which the secret token method has already been implemented. Thus, these are mostly manageable problems, though not as much so as SQL injections.

Cross-Side Scripting (XSS). Similarly to CSRF, this attack involves a user visiting a good server and a malicious server: it visits the malicious server first, and gets a malicious link which adds malicious Javascript to the code. For example, one might have a link such as `http://victim.com/search.php?term=<script>`*bad stuff...* `</script>`, which, as you can imagine, can do lots of malicious stuff. Then, the victimized server echoes the user input back to the victimized client, which is clearly not a good thing.

An XSS vulnerability is present when the attacker can inject code into a session that isn't its own. This can also happen in email forms, with the same kind of attack (e.g. a PayPal attack, where the link led to a PayPal site that said their email was compromised, but that led to a phishing site).

So the victim can see the URL before clicking, but really, who does? And the link can look very innocuous, but actually leads somewhere else. So that doesn't even make much of a difference.

Sometimes this gets weirder: Adobe Acrobat has a Javascript interpreter, so feeding it a URL that has Javascript in it leads to a script being executed, and it can read local files. So there are lots of ways that scripts can get into one's system and monkey around; this is the hardest kind of attack to defend against, and there's no single silver bullet to get around it.

Another example is for an attack server to inject malicious scripts into a victim server, and then when the user visits the victim server, it ends up sending data to the attacker!

For example, do you remember MySpace? Me neither. Anyways, users can post HTML on their pages, so one guy set up some Javascript that made anyone who visited become his friend... and spread it to their friends. Then, he had millions of friends in 24 hours, which caused MySpace to go offline for a little bit.

One can also used DOM-based XSS, without even dealing with a server. Imagine a script which calls a welcome page with the script `<script>alert(document.cookie)</script>`. In other words, any time there's a call to rendering in a page, the attacker can ask, "how do I put a script in there, and what do I do with it?"

Normally, XSS is considered a vulnerability at the victim site. It's about input/output checking: if you ever receive a script or something that looks like it, turn it into a string that's not a script and remove it, and whatever you do, don't send it back out! For example, in `ASP.NET`, one can set a flag for pages that should have no scripts in them. This can get complicated, filters don't work very well, etc. And taint analysis or static analysis can be used to make this work as well.

In other words, we need relatively sophisticated tools to defend against these broader and more complicated attacks.

But as a client, you can do something, too! (Or a browser designer.) The browser developer can (and the IE developers did) implement a feature where they check outgoing posts and remove scripts that reflect user input (and send data elsewhere). The vast majority of the time, this works, and has perhaps a few false positives. However, it prevents frame-busting as we talked about last time, which is a useful thing to have. It's nonetheless a great example of how complicated web security really is, and everyone's attempts to do well step on each others' toes.

Because of the importance and complexity of web security vulnerabilities, there are a lot of businesses that check for vulnerabilities; they can't find all of them, but manage to find a significant number of web vulnerabilities (though some companies do a lot better than others). And automated tools also help for many simple vulnerabilities.