Episode 2 – OS Story 2: Security

Basics of computer architecture

Processes run on a CPU. The CPU 'calls the shots' – it can access memory, write to disks, display things on screen, etc.

In principle (and certainly, in the 1980/1990s when using DOS), if there is no protection in place, any process running on the CPU could access any hardware. There are machine code instructions for e.g. writing to IO devices that any program can use. Predictably, there were some quite spectacular viruses that took advantage of unrestricted access to the machine.

Kernel mode, user mode

With modern operating systems, there is thankfully some protection. Ordinary processes run in **user mode** (level 1). These cannot write to IO devices, or access memory that isn't reachable via their page table.

The kernel – the heart of the operating system that actually does memory management, provides device drivers, etc. etc. – runs in **kernel mode** (level 0). This can do whatever it likes, including the **privileged instructions** on the CPU for performing IO.

If our process actually wants to do IO, or request more memory, it makes a **system call** or **software interrupt**. This allows it to specify a task for the kernel to perform on its behalf. It triggers a switch from user mode to kernel mode; then back. The interrupt is processed by the **interrupt handler**.

The **mode bit** for the system stores whether it is in kernel mode, or user mode, at the current moment. The value of the mode bit is checked by the CPU every time there is a privileged instruction. Thus, if a programmer tries to sneak a privileged instruction into their process – which is running in user mode – it won't work, the process will instead be killed.

System calls

System calls are a lot like methods. They have a name, and take some parameters.

The name, internally, maps to a number. e.g. on Linux, the system call 'open' is number 5, write is number 4, ...

To make an interrupt, the process puts the system call number, and the parameters, into registers; then runs 'syscall' – which makes the interrupt happen. The operating system then takes over, runs the interrupt handler for the given system call number, and when this completes, returns to the process and carries on from when it left off.

There is a caveat: if the system call has lots of parameters (more than there are registers in the CPU), one needs to use the stack, or a block of memory, to pass the parameters to the system call.

Direct Memory Access

The hardware of the machine can access memory itself. For instance, to read 128Kb from a file on disk:

- A process requests a 128Kb block of memory from the operating system, to use as a buffer;
- It then makes the system call 'read', requesting 128Kb of data is read from some file into this buffer.
- The hard disk will go ahead and then do this transfer, sending the data straight into the buffer.
- Once this has finished, the process is resumed (it moves off the 'waiting for IO' queue in the scheduler, back onto the 'ready' queue)

To stop horrible mess-ups when using virtual memory and DMA, any pages being used for DMA (in this example, the 128Kb of memory the hard drive is writing to) are protected by an **IO interlock** and cannot be chosen as victims.

To keep the number of pages that have IO interlocks to a minimum, some processes use a trick: they request more than 128Kb of memory, then find a 128Kb region within that that is **aligned** exactly with a page boundary. That is, the 128Kb buffer starts at some memory address is (p,0): on a page p, with offset 0. If the system has pages of size 4Kb that then means it is using 128/4 = 32 pages; rather than maybe 33 pages if the 128Kb block of memory started half way through one page, and finished half way through another.

Kernel architectures

Monolithic kernels: system calls stay in kernel mode.

Microkernels: move some kernel code back out into user mode again. For instance, when swapping a page out to disk, the actual swapping will need to be in kernel mode (to be able to read/write from/to disk), but the algorithm that chooses the victim page? Maybe not.

The caveat is that context switching from user mode to kernel mode takes time. Hybrid kernels, such as XNU (the basis for the kernel on OS X) keep some code in kernel mode that doesn't strictly need to be, just to speed things up a bit.

Stack overflows

Back to page 1 – the stack is used for temporary variables whilst a program is running.

The stack grows downwards. How far down it has currently reached is denoted by the **stack pointer.** If e.g. a 16 byte array is pushed onto the stack, the stack pointer goes down by 16 bytes. When that 16 byte array is no longer needed, the stack pointer goes up by 16 bytes again.

When a function is called:

- The **return address** is pushed onto the stack – with reference to slide 20, when calling getPassword() at line 2, the return address would be line 3. This is pushed onto the stack, decreasing the stack pointer by e.g. 8 bytes, to store this.

- Execution then jumps to the function itself, and the stack pointer is decreased according to how much space is needed for the temporary variables of that function: e.g. for getPassword on slide 19, at least 9 bytes is needed for the buffer 'buf' so the stack pointer will be decreased by 9.
- When the function completes, the space for its temporary variables is released (in the example, 9 bytes); the return address is taken off the stack; and execution jumps back to there.

A stack overflow attack exploits the fact that the temporary data of a function is stored right before the return address. If the program is taking a string as input (e.g. using gets()) and one types in too many bytes (enough to overflow a fixed-size buffer), then one can overwrite the a return address. Then, at the end of the function, execution will jump to there, rather than where it really should have gone.

For the example on the slides, pressing space 9 times would fill the buffer; then entering 4 would replace the legitimate return address of 3 with 4. When getPassword() then finished, execution would jump to line 4: bypassing the check on line 3 that the correct password was entered.

In reality, this occurs in machine code, not in C code. e.g. If you have a compiled executable on disk, e.g. myprogram, then at a terminal on Linux:

```
objdump -d myprogram
```

...will show you what the machine code is for that program. Overleaf is the result of running this on a compiled version of the C program on the lecture slides. The key details are:

- In <main> the function getPassword is called at the line that contains **callq.** This pushes the return address onto the stack – the address of the next line of code, which is `4005d8`.
- At the start of 'getPassword' the stack pointer is decreased by 0x10 bytes: this is hexadecimal. 0x10 = 16 in decimal.
- If we then enter 16 space characters, that will fill the buffer; anything after that will overwrite what was on the stack after the buffer – the return address. If we then enter the number `4005e1` this will replace the return address on the stack with 'line 4' from the C code: the point at which 'giveThemSuperPowers' is called.

One thing to remember is that on 'x86' machines (those with Intel/AMD CPUs etc.) memory addresses are stored in **little endian** format. Thus to go to 4005e1 we would enter the bytes

e1 05 40

..i.e. start at the right, and enter two hexadecimal digits at a time.

```
0000000000400586 <_Z11getPasswordv>:
  400586:       53                      push   %rbx
  400587:       48 83 ec 10             sub    $0x10,%rsp
  40058b:       48 89 e7                mov    %rsp,%rdi
  40058e:       e8 ed fe ff ff          callq  400480 <gets@plt>
  400593:       bf 74 06 40 00          mov    $0x400674,%edi
  400598:       b9 09 00 00 00          mov    $0x9,%ecx
  40059d:       48 89 e6                mov    %rsp,%rsi
  4005a0:       f3 a6                   repz cmpsb %es:(%rdi),%ds:(%rsi)
  4005a2:       0f 97 c2                seta   %dl
  4005a5:       0f 92 c0                setb   %al
  4005a8:       38 c2                   cmp    %al,%dl
  4005aa:       75 0a                   jne    4005b6 <_Z11getPasswordv+0x30>
  4005ac:       c7 05 96 0a 20 00 01    movl   $0x1,0x200a96(%rip)     # 60104c <isRoot>
  4005b3:       00 00 00
  4005b6:       48 83 c4 10             add    $0x10,%rsp
  4005ba:       5b                      pop    %rbx
  4005bb:       c3                      retq

00000000004005bc <_Z19giveThemSuperPowersv>:
  4005bc:       48 83 ec 08             sub    $0x8,%rsp
  4005c0:       bf 7d 06 40 00          mov    $0x40067d,%edi
  4005c5:       e8 86 fe ff ff          callq  400450 <puts@plt>
  4005ca:       48 83 c4 08             add    $0x8,%rsp
  4005ce:       c3                      retq

00000000004005cf <main>:
  4005cf:       48 83 ec 08             sub    $0x8,%rsp
  4005d3:       e8 ae ff ff ff          callq  400586 <_Z11getPasswordv>
  4005d8:       83 3d 6d 0a 20 00 00    cmpl   $0x0,0x200a6d(%rip)     # 60104c <isRoot>
  4005df:       74 05                   je     4005e6 <main+0x17>
  4005e1:       e8 d6 ff ff ff          callq  4005bc <_Z19giveThemSuperPowersv>
  4005e6:       b8 00 00 00 00          mov    $0x0,%eax
  4005eb:       48 83 c4 08             add    $0x8,%rsp
  4005ef:       c3                      retq
```

Stack protection

A **canary word** can be put on the stack at the **start** of the function. This will then sit between the buffer, and the return address.

Overflowing the buffer will then overwrite the canary word, before overwriting the return address.

If the canary word is checked at the **end** of the function, and it is not equal to what it should be (i.e. what was put on the stack at the start), then the program is terminated.

Modern compilers will automatically insert canary words into the stack at the start of functions, and check them at the end.

Other protection

The code on the previous slide has memory addresses written on it. A stack overflow attack relies on these being predictable – we have to know what value to overwrite the return address with.

Library load order randomisation loads libraries of shared code in a random order. They're still next to each other in memory, but attacks are less likely to succeed – if there are enough libraries, at least.

Address space layout randomisation completely randomises the addresses where libraries etc. are loaded – they're no longer next to each other. This makes it even more unlikely that attacks will succeed, unless there is some way of persuading the process to leak the memory addresses of where the libraries are.

Access control via users and groups

In theory we want to be able to say for each user on the machine which files they have access to.

In practice, this would require a lot of data to be stored. To make it more efficient users are aggregated into **groups**. File permissions usually then say what the owner can do with the file (read/write/execute), what the users in the group for the file can do, and what any other user can do.

Using an **access control list** one can have multiple groups attached to a file, which increases flexibility. The week 8 quiz demonstrated some instances where an access control list would be needed.

Because some system programs need administrator access (e.g. changing passwords) it is possible to provide a mechanism for **privilege escalation** – if the owner of an executable file sets the UID bit on that file, then when that program is running, it can access any files the owner can access.

But, it is a bad idea to set the UID bit on large programs, as that opens up the potential for exploiting bugs in those programs for malicious purposes. Use clients and mechanisms instead, with a small mechanism doing the privileged operation; and the rest of the code as a client.

Hash functions

Plain-text password goes in; encrypted password comes out.

If it's easy to go from the encrypted password to the original, you're doing it wrong.

Getting passwords

Breaking the maths behind hash functions is hard. Other options are more likely to succeed:

- Brute-forcing passwords is in itself not so successful unless the password is really short (though hey, passwords often are too short, so it can sometimes work). If one has the hashed version of the password, a **rainbow table** can be used in some cases: it lists precomputed hashed values for known passwords, which is quicker to work through than having to run the hash function on possible passwords (the maths has already been done).
- Social engineering often works. Ask people nicely; use the names of their pets; etc.
- The maths behind cryptography might be sound; but implementations of cryptographic software can contain errors.

Heartbleed

Heartbleed was a pretty big-news vulnerability. To secure data for transmission over the internet, **public—private key pairs** are often used. The server keeps the private key; the client uses the server's public key; and then only the server can read what was sent to the client.

Heartbleed allowed users to fish 32Kb of data at a time from the server – which if they were lucky, included a private key. Repeat it often enough, and the odds of it working increases.

MiFare security cards

Hardware can still contain bugs. The cryptography on MiFare Classic cards is compromised – there's a bug in the hardware that means the keys for the card can be extracted in 30 seconds, with just the card, and a cheapo card reader bought from the Internet.

Oyster no longer uses MiFare Classic, it uses MiFare DESFire. But it did used to be possible in principle to illegally top up ones own Oyster card, for no money.

SQL Injection

As in the week 8 coursework: one can enter a password that contains a fragment of SQL. If the server is then doing String operations to make the SQL query, and is not careful about how it does this, then this fragment of SQL will be included in the statement – allowing password protection to be bypassed.

Accessing APIs

See the slides – your APIs really must handle sessions properly.

Email security

Email is by default insecure – it's sent in plaintext over untrusted networks, with no way by default of the sender proving they are who you think they are.

Thus, bellogate – it's possible to impersonate people.

Email security does exist, e.g. encrypting emails, but it's an option, not the default.

Cross-site scripting (XSS)

If user inputs can make it into your web pages, without having HTML tags removed, that's a problem. It means if e.g. the user can enter a product review, or post something on KEATS, they can write some Javascript that will be included on the webpage.

Stored XSS is where their Javascript is stored by your server, and sent to visiting users.

Reflected XSS is where it's possible to sneak some Javascript into a URL; the attacker then gives that URL to someone, who clicks on it, and is served a page containing the malicious Javascript. But at least it's not actually being stored on your server – it's just reflected back to whomever visits that link.

One use for XSS is to get the session keys (e.g. document cookies) that authenticate user sessions. For instance, reflected XSS could be used to get a user's cookie and send that to the attacker's server.

Arbitrary code execution

If you take user input, and that's used to make commands, that can go horribly wrong unless you can absolutely guarantee there's no way the user input itself could be taken as a command. Cue the bash vulnerability: by judicious use of a semicolon (which means 'end of command') the user could then specify arbitrary code that would be ran on your server.

Physical access to the hardware

Much harder to protect systems that potentially malicious users have access to. It's a bigger deal than it used to be, as tricks such as packet sniffing work on unsecure wireless networks too – e.g. in coffee shops or airports.

Trusting other people's networks

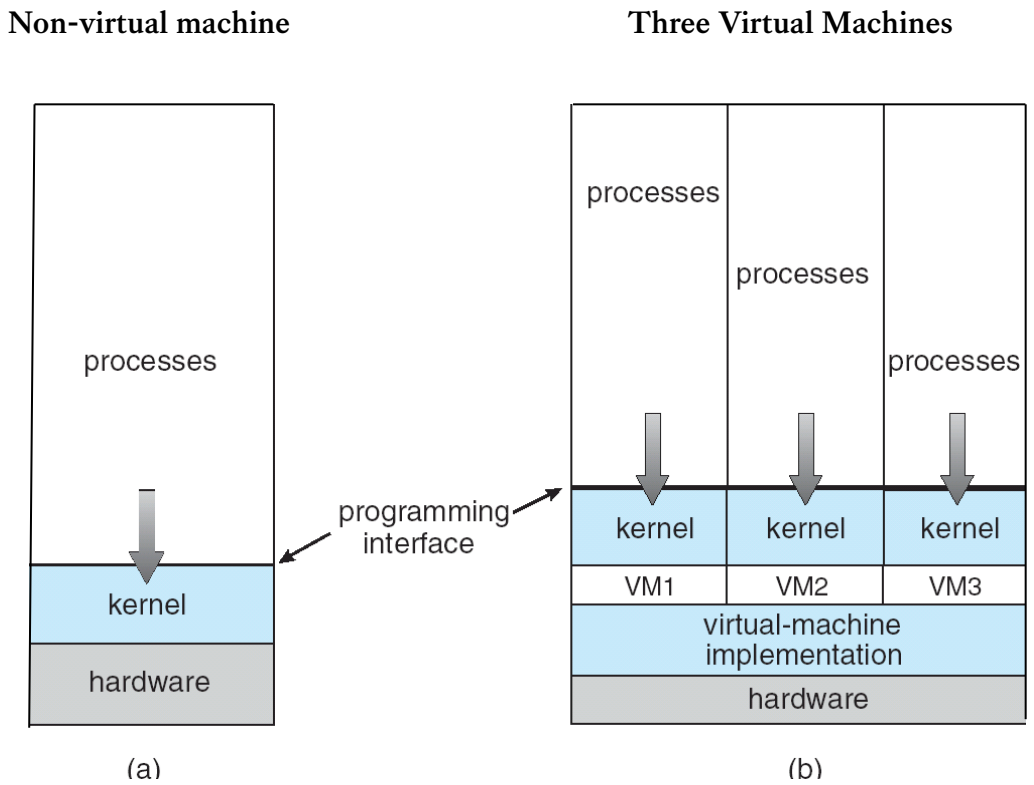
Then again, when using other people's networks, their servers might be performing **man in the middle** attacks, intercepting your connections to the Internet. They might even send you forged private keys. Your browser will notice, but maybe the average user wouldn't notice the warning, or would just click OK. Don't be the average user....

Episode 3 – Linux (a.k.a. the Penguin of Azkaban)

Virtual machines

Virtual machines allow one OS to be ran on top of another: a **guest OS** on top of a **host OS**. They intercept attempts by the guest OS to access hardware and simulate what would happen. For instance, a virtual hard drive is just a big file on disk on the host OS; a virtual graphics card sends pixels to the window in which the guest OS is running, rather than to the actual screen; and so on.

It's a bit like this:



VMs are useful for various reasons. For a developer, they allow several OS installations to run on one machine, for testing purposes (e.g. running Windows and Linux on a Mac laptop). From a sysadmin point of view, they're useful for managing customers – use one VM for each customer's server. Then, each customer can have root access just to their VM, without needing root access to the original machine; an attack on one customer's installation due to their ineptitude is less likely to cause problems for the other servers running on that machine; they can have their memory usage limited (e.g. the VM gets 4GB of memory) ensuring again they can't disrupt the other customers' installations.

VMs were covered in this lecture as to a point, they had to go somewhere, and they're really useful to know about. Plus in any case, if you're unfortunate enough not to be running Linux, you can install it on a VM and use it there.

Linux process basics

Processes have an owner (see Access Control in the last lecture). Any processes spawned by this process, inherit that owner. Processes also have a parent, so e.g. a process' child processes will be terminated when it itself is terminated. This is useful to avoid zombie processes, that were started, and are still running, but that are waiting for data from a parent that no longer exists.

Processes also have a priority (on Linux, a 'niceness'). More on that later.

A shell

Short answer: A glorified way of starting processes by typing in the name of which one to run, and the arguments to give it.

Longer answer: Shells actually do more than that – they provide a syntax for saying what should be done with the input or output to/from a process. By default, for a shell running in a terminal (a 'command prompt') **standard input** is whatever is typed at the keyboard; **standard output** goes to the screen; and **standard error** goes to the screen, too. But, we can change this:

- Pipes connect standard output of one process to standard input of another. (They can connect standard error to standard input too, but this is much more rare.) This allows us to combine lots of small processes, to do large, interesting tasks.
- Redirection either changes standard input to read from a file on disk; or changes standard output/error to be sent to a file on disk.

Pipes use the | character, redirection uses > to send to a file or < to read from a file.

Virtual filesystems

I mentioned two virtual filesystems:

- /proc/... presents information about processes running on a machine, as if they were files on disk. You can browse these to see what's going on behind the scenes.
- /dev/... contains **block devices**, for accessing the hardware of the machine. These will often require root access to prevent the user from filling the disk with zero bytes. But, they do reduce hardware access, to file-access, which is very useful – e.g. reading live video from a webcam, is like reading video from an infinitely long file on disk.
    - /dev/random – reading from this produces random bytes
    - /dev/zero – reading from this produces ASCII-value-zero bytes
    - /dev/null – writing to this sends output to absolutely nowhere.
        - '`something > /dev/null`' means 'run `something`, throw away its output'.

Buffering

IO is generally buffered. This avoids making excessive system calls, avoids having to wait for the network each time a small amount of data is sent, etc. etc.

Sometimes buffering is annoying, so it can be turned off. Standard error has no buffering by default, so that error messages appear right away. Standard output is usually buffered.

Security

Obligatory security paragraph: running `ps aux` shows which processes are running and what command-line arguments they were given. Be aware of this.

Process signals

There are lots of ways of killing a process, some are kinder than others. **KILL** is the most brutal, immediately killing the process. **TERM** is the default, giving it chance to close its open files, disconnect users nicely, etc. **HUP** is sent by terminals when they are closed. If you want to keep a server running after you've started it at some terminal, use `nohup`: this causes it to ignore the HUP signal, and hence carry on running in the background. (nohup is usually used in conjunction with `&` at the end of the command line.)

Be nice

A nice process (nice=20) only uses spare CPU cycles. A not-nice process (nice=-20) will hog the CPU given half a chance. By default processes run with nice=0. Only root can make processes less nice; you can make your own processes more nice if you want.

The environment

The environment is a variable—value hash. Environment variables store all sorts of useful things, e.g. the paths to where the shell should look for binary (executable) files on disk; the current working directory; the language; the user's home directory; etc. etc. Run `export` at a bash prompt to see these.

Specific commands

There are a lot of commands on Linux. I don't expect you to memorise what they are, but you should be broadly aware of the general ideas – i.e. that there are lots of commands to do lots of little things, and we can uses pipes and redirection with them to do larger, more-impressive things. If I had to pick a keyword, remember that a **filter** is a command that reads from standard input and writes to standard output. A lot of the examples from the lecture (grep, sed, wc) are filters, and can be combined together in useful ways.

Lecture 11

Unless it's covered elsewhere on the course (e.g. how to make threads in Java), none of the content on this lecture is on the exam.