Post_Mortems

Hindsight is 20-20 - Unknown

This chapter is meant to serve as a big “why are we learning all of this”. In all of your previous classes, you were learning what to do. How to program a data structure, how to code a for loop, how to prove something. This is the first class that is largely focused on what *not* to do. As a result we draw experience from our past in very real ways. Sit back and scroll through this chapter as we tell you about problems of programmers past. Even if you are dealing in something much higher level like web-development, everything relates back to the system.

Shell Shock

Required: Appendix/Shell

This was a back door into most shells. The bug allowed an attacher to exploit an environment variable to execute arbitrary code.

```
> env x='() { :;}; echo vulnerable' bash -c "echo this is a test"
vulnerable...
```

This meant that in any system that uses environment variables and doesn’t sanitize their input (hint no one sanitized environment variable input because they saw it as safe) you can execute whatever code you want on others machines including setting up a web server.

Lessons Learned: On production machines make sure that there is a minimal operating system (something like BusyBox with DietLibc) so that you can understand most of the code in the systems and their effectiveness. Put in multiple layers of abstraction and checks to make sure that data isn't leaked. For example the above is a problem insofar as getting information back to the attackers if it is allowed to communicate with them. This means that you can harden your machine ports by disallowing connections on all but a few ports. In addition, you can harden your system to never perform exec calls to perform tasks (i.e. perform an exec call to update a value) and instead do it in C or your favorite programming language of choice. Although you don’t have flexibility, you have peace of mind what you allow users to do.

Heartbleed

Required: Intro to C

To put it simply, there was no bounds on buffer checking. The SSL Heartbeat is super simple. A server sends a string of a certain length, and the second server is supposed to send the string of the length back. The problem is someone can maliciously change the size of the request to larger than what they sent (i.e. send “cat” but request 500 bytes) and get crucial information like passwords, SSL keys etc from the server. There is a https://xkcd.com/1354/
(https://xkcd.com/1354/)

Lessons Learned: Check your buffers! Know the difference between a buffer and a string.

Dirty Cow

Required: Processes/Virtual Memory

https://en.wikipedia.org/wiki/Dirty_COW
(https://en.wikipedia.org/wiki/Dirty_COW)

A process usually has access to a set of read-only mappings of memory that if they try to write to they get a segfault. Dirty COW is a vulnerability where a bunch of threads try to access the same piece of memory at the same time hoping that one of the threads flips the NX bit and the writable bit. After that, an attacker can modify the page. This can be done to the effective user id bit and the process can pretend it was running as root and spawn a root shell, allowing access to the system from a normal shell. Naturally this is undesirable.

Lessons Learned: Spinlocks in the kernel are hard.

Meltdown

There is an example of this in the background section

Spectre

Same here

Mars Pathfinder

Required sections: Synchronization and a bit of Scheduling

https://www.microsoft.com/en-us/research/people/mbj/#!just-for-fun (https://www.microsoft.com/en-us/research/people/mbj/#!just-for-fun)

The mars pathfinder was a mission that tried to collect climate data on Mars. The finder uses a single bus to communicate with different parts. Since this was 1997, the hardware itself didn't have advanced features like efficient locking so it was up to the operating system developers to regulate that with mutexes. The architecture was pretty simple. There was a thread that controlled data along the information bus, communications thread, and data collection thread in with high, regular, and low priorities with respect to scheduling. The other caveat is that if an interrupt happened at some interval and a task is running and a task is to be scheduled, the task that has the higher priority wins.

The pattern that caused everything to start failing was the data collection thread starts writing to the bus, the information bus thread is waiting on the data and then out of nowhere the communication thread comes in. The communication thread would preempt the other lower priority thread **while the lower priority thread still held the mutex**. This means when the regular priority thread tried to lock the bus, you'd get deadlock. After some time the system would reset, but ideally this isn't a good thing to leave to chance.

Moral of the lesson? Don't have the applications themselves deal with the synchronization. Define a module that handles mutex locking and have module communicate with each others through files, IPC, etc etc.

Mars Again

Required Sections: Malloc

https://www.computerworld.com/article/2574759/data-storage-solutions/out-of-memory-problem-caused-mars-rover-s-glitch.html (https://www.computerworld.com/article/2574759/data-storage-solutions/out-of-memory-problem-caused-mars-rover-s-glitch.html)

The short of it is they ran out of memory. The long of it is they ran out of memory, disk space, and swap space. The moral of the story? Make sure to write code that can handle file failures and can handle files when they close and go out of memory, so the operating system can hotswap file to free up memory. Also clean up files, assume that your tmp directory is roughly a hundredth or a thousandth of the total size and use that.

Year 2038

Required sections: Intro to C

https://en.wikipedia.org/wiki/Year_2038_problem (https://en.wikipedia.org/wiki/Year_2038_problem)

This is actual a fundamental problem that hasn't happened yet. Unix time stamps are kept as the number of seconds from a particular day (Jan 1st 1970). This is stored as a 32 bit signed integer. In March of 2038, this number will overflow. This isn't a problem for most modern operating system who store 64 bit signed integers which is enough to keep us going until the end of time, but it is a problem for embedded devices that we can't change the internal hardware to. Stay tuned to see what happens.

Lessons learned: Plan like your application will be huge one day.

Northeast Blackout of 2003

Required Sections: Synchronization

https://en.wikipedia.org/wiki/Northeast_blackout_of_2003 (https://en.wikipedia.org/wiki/Northeast_blackout_of_2003)

Very simply put a race condition triggered a series of undefined events in a system that cause the blackout of most of the northeastern part of North America for quite some time. This bug also turned off or caused the backup system and the logging systems to fail so people didn't even know of the bug for an hour. The exact bits that were flipped are not known, but patches have been put into place.

Lessons Learned: Modularize your code to localize failures (i.e. keeping race conditions different between processes). If you need to synchronize among processes make sure your failure detection system is not interlaced with your system.

Apple IOS Unicode Handling

Required Sections: Intro to C

http://appleinsider.com/articles/15/05/26/bug-in-ios-notifications-handling-crashes-iphones-with-a-simple-text (http://appleinsider.com/articles/15/05/26/bug-in-ios-notifications-handling-crashes-iphones-with-a-simple-text)

Wonder why we teach string parsing? Because it is hard thing to do even for professional software developers. This bug allowed a lot of undefined behavior when trying to parse a series of unicode characters. Apple probably knows why this happened, but my guess is that the parsing of the string happens somewhere inside the kernel and a segfault is reached. When you get a segfault in the kernel, your kernel panics, and the entire device reboots. Undefined behavior means anything though, and a lot of varied things did happen with this bug.

Lessons Learned: Fuzz your kernel

Apple SSL Verification

Required Sections: Intro to C

https://en.wikipedia.org/wiki/Unreachable_code#Examples (https://en.wikipedia.org/wiki/Unreachable_code#Examples)

Due to a stray goto in Apple's code, a function always returned that an SSL certificate was valid. Naturally hackers were able to get away with some pretty crazy site names.

Lessons Learned: Always bracket if statements, use gotos sparingly. Chances are if you need to use a goto, write another function or a switch statement with fall throughs (still bad).

Sony Rootkit Installation

Required Sections: Intro to C/Processes

https://en.wikipedia.org/wiki/Sony_BMG_copy_protection_rootkit_scandal (https://en.wikipedia.org/wiki/Sony_BMG_copy_protection_rootkit_scandal)

Picture this. It's 2005, limewire came a few years prior, the internet was a growing pool of illegal activities – not to say that is fixed now. In this day, Sony knew that it didn't have the computing power to police all the interwebs or get around the various technologies people were using to get around copyright protections. So what did they do? With 22 million Music CDs, they required users to install a rootkit on their operating system, so Sony can monitor the device for unethical activities.

Privacy concerns aside, and believe me there are a lot of them, the big problem was that this rootkit is a backdoor for everyone's systems if not programmed correctly. A rootkit is a piece of code usually installed kernel-side that keeps track of almost anything that a user does. What websites visited, what clicks or keys typed etc etc. If a hacker finds out about this and there is a way to access that API from the user space level, that means any program has the ability to find out important information about your device. Needless to say, people were not pleased.

Lessons Learned: Get an antivirus and/or apparmor and make sure that an application is only requesting permissions that make sense. If you are really torn, try something like Windows sandbox or keep a Sacrificial VM around to see if installing it makes your computer horrible. Don't trust certificates trust code.

Civilization and Ghandi

Required Sections: Intro to C

https://www.geek.com/games/why-gandhi-is-always-a-warmongering-jerk-in-civilization-1608515/ (https://www.geek.com/games/why-gandhi-is-always-a-warmongering-jerk-in-civilization-1608515/)

This is probably well known to gamers why someone as (in real life) non-violent as Ghandi was very aggressive in the video game civilization. In the original, the game kept aggressiveness as an unsigned integer. During the course of the game the integer could be decremented and then the problem ensued because Ghandi was already at 0. This caused him to become the most aggressive character in the game.

Lessons Learned: The take away from this is **Never** use unsigned numbers unless you have an express written reason for it (reasons include you need to know about the overflow behavior, you are bit shifting, you are bit masking). In every other case, cast it.

The Woes of Shell Scripting

Required Sections: Intro to C/Appending

https://www.pcworld.com/article/2871653/scary-steam-for-linux-bug-erases-all-the-personal-files-on-your-pc.html (https://www.pcworld.com/article/2871653/scary-steam-for-linux-bug-erases-all-the-personal-files-on-your-pc.html)

There was a simple bug in steam that caused steam to remove all of your files in the form of something like this

```
> ROOT=$(cd $0/; echo $PWD);
> rm -rf $ROOT
```

Well what happens if $0 or the first parameter passed into a script doesn't exist? You move to root, and you delete your entire computer.

Lessons Learned: Do parameter checks, always always always set `set -e` on a script and if you expect a command to fail, explicitly list it. You can also alias rm to mv and then delete the trash later.

Appnexus Double Free

Required Sections: Intro to C/Malloc

https://techblog.appnexus.com/2013-09-17-outage-postmortem-586b19ae4307 (https://techblog.appnexus.com/2013-09-17-outage-postmortem-586b19ae4307)

It may not be as simple as it sounds. Appnexus uses an asynchronous garbage collector that reclaims different parts of the heap as it believes that objects are not being used anymore. The architecture is that an element is in the used list and then it is taken out to a to-be-freed list. After a certain time if that element was not used, it is freed and added to the free list. This is fine until two thread try to delete the same object at once, adding to the list twice. After less time, one of the objects was deleted, the delete was announced to other computers.

Lessons Learned: Try not to make hacky software if you need to. Modularize, and set memory limits, and monitor different parts of your code and optimize by hand. There is no general catch-all garbage collector that fits everyone. Even highly tested ones like the JVM need some nudges if you want to get performance out of them.

ATT Cascading Failures - 1990

Required Sections: Intro to C

http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html (http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html)

To be honest the explanation of the bug is really good at the link above, I recommend reading that. Basically it was a series of network delays that caused some switches (telephone) across the country to think that other switches were operable when they weren't. When the switches came back online, they realized they had a huge backlog of calls to route and began doing so. Other routing failures and restarts only compounded the problem.

Lessons Learned: Not using C would've actually helped here because of more rigorous fuzzing (though C++ in this day and age would be worse with its language constructs). The real moral of the story is networks are random and expect any jump at any point in your code. That means writing simulations and running them with random delays to figure out bugs before they happen.