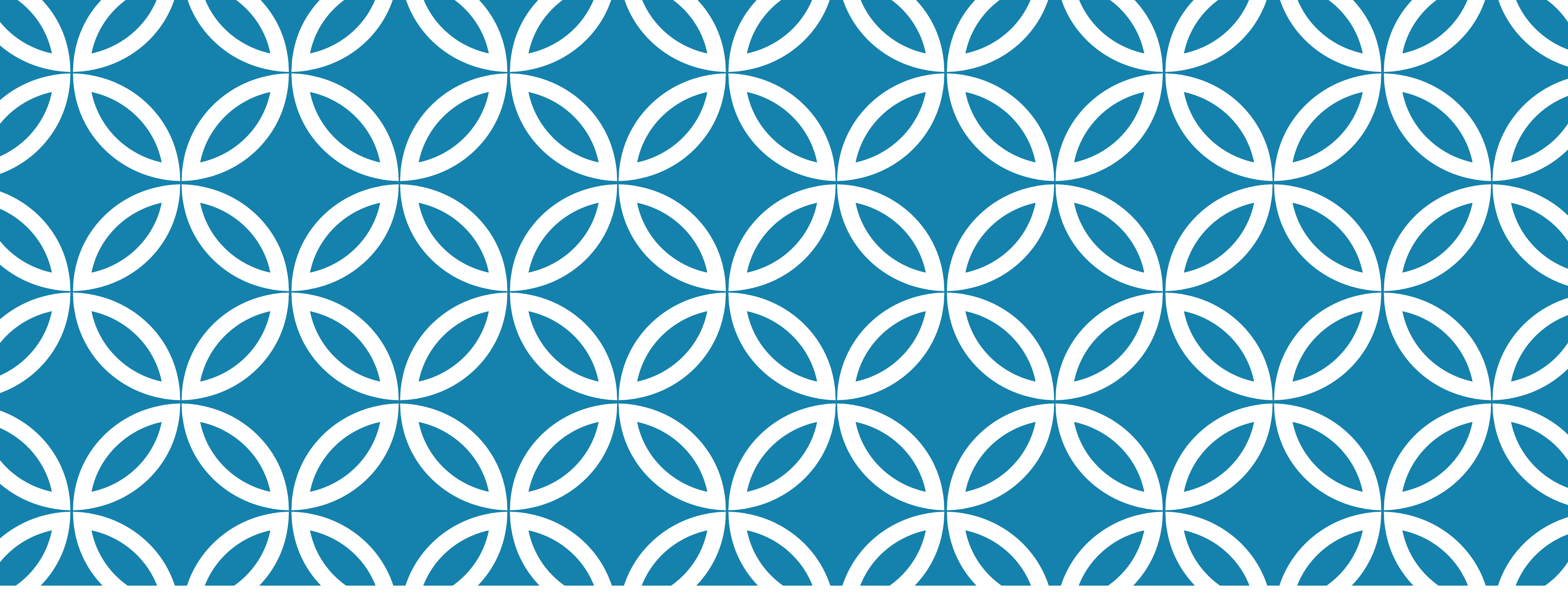


LINUX / C++ PROTECTION FEATURES

Professor Ken Birman
CS4414 Lecture 27

IDEA MAP FOR TODAY

It's a (cyber)war out there!

Firewalls and Memory Protection

Type Checking as a Protection Tool

Concept of Defense in Depth

We actually *can* protect valuable systems!

HACKING: WHAT HAVE WE LEARNED?

… by 1988, Unix was a terrible mess riddled with holes! Linux emerged in 1991 but inherited many of the same issues.

In fact in the subsequent 31 years, many have been fixed. But nobody doubts that many remain!

Today, there is far more emphasis on hardening these platforms against exploits of all kinds.

CODE AND PLATFORM REVIEWS

Companies are getting contracts to review the code for Unix, Linux and major applications.

Many work with their own tools, and apply them to the code base to search for risky business. Then they report the issues as potential bugs.

There are companies that maintain Linux, and they fix the bugs. Unix is a legacy system and no longer maintained, or in wide use.

MODERN LINUX

Every single use of memcpy and strcpy and similar functions has been extensively checked.

This should have reduced the risk of buffer overrun attacks substantially.

Tools (similar to Valgrind) exist that do automated checks for unsafe copying, and have been used on Linux by professionals.

MODERN LINUX

All APIs have been scrutinized too, by red teams

- These are groups funded to try and find a flaw
- Often they include people who were previously black-hat hackers but were caught, or perhaps switched to the good side.

This includes every single "privileged" application, within the standard Linux distributions.

EVERYONE IS SURE THAT BUGS REMAIN

It is particularly hard to check Linux for bugs.

One concrete issue is that Linux is coded in C, which has pointers, threads, shared memory, interrupts, etc. These features leave many opportunities for subtle race conditions and other errors.

Sophisticated hackers sometimes find such issues, then find exploits that somehow target them.

IT ISN'T BEING REWRITTEN IN RUST

Changing to a different language might be helpful, but Linux loads modules into memory dynamically and does other things that are security risks, too.

So the bottom line is that rewriting it in Rust wouldn't change things very much. And doing that probably would slow Linux down. A full rewrite in C++ is a more likely thing, someday.

IDEAL WORLD?

In languages that enable very rich specifications for modules and code, we can use "formal prover" tools to go much further

For each method, we arrive at invariants about the situations in which it would run, and that it must "reestablish" after executing.

Then the developer works to prove that the methods satisfy these properties, using the **theorem provers.**

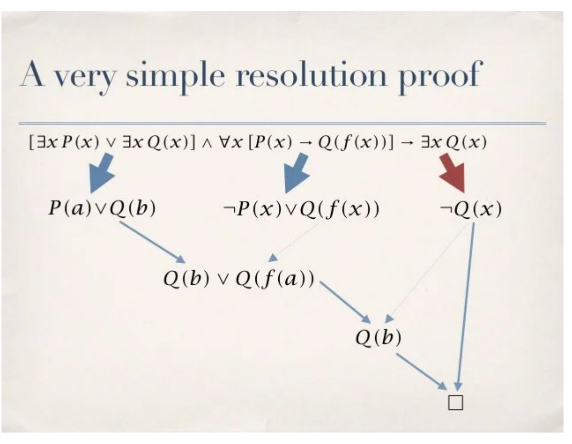
WHY “THEOREM PROVERS”?

The idea is to use automated tools to actually prove things like memory safety, and to run them before we compile the code.

There is a big push to enable this for C++, too!

In ten or fifteen years we may literally have compile-time memory safety as part of what C++ can support!

C-CURED, RUST, SEL4



Some languages are much more strongly checkable. We learned about Rust in Lecture 27. C-Cured is another famous example. They bring significant costs but are easier to formalize and reason about.

People have created versions of Linux using these languages, and there is even a proved-correct C compiler! And a proved-correct microkernel (definition: a bare minimum OS). It is called SEL4.

But Linux is used in a million ways and is huge and complex. Many features are omitted in SEL4, although it is great for things like nuclear reactor control systems and airplanes with fly-by-wire flight management.

BUT…

The languages in which proving is most successful are often very heavily type-checked in ways that preclude the kinds of high-efficiency logic we've explored in CS4414.

… it would be nice if this could change, and over time, it will.

Today, C and C++ are very far from being verifiable in this sense.

TYPE CHECKING, MEMORY PROTECTION

Modern systems deal with a tradeoff

We can harden them by doing aggressive type checking and using restrictions on what individual segments of memory can contain and how they can be used.

But these steps harm performance

IS THERE HOPE? DEFINITELY!!!

If you find yourself working on a safety-critical application, you can and should consider these proved correct packages.

For general purposes, progress has been slower. Over the span of years we are definitely seeing coverage expand (not quickly).

IS THERE HOPE? DEFINITELY!!!

If you find yourself working on a safety-critical application, you can and should consider these proved correct packages.

For general purposes, progress has been slower. Over the span of years we are definitely seeing coverage expand (not quickly).

Key insight? Your system *will* be under attack. No program is bullet-proof. So, anticipate issues and build in self-checks that can detect and repair compromised elements. Like fault-tolerance.

YOU NEED TO PROTECT THE HARDWARE, THE PLATFORM AND THE APPLICATIONS

Imagine that you have been hired to look into a rash of burglaries.

You visit and discover that none of the homes had locks on the doors. You recommend locks.

The next year you visit again… the problem is just as bad! Now the crooks are climbing up to the second floor windows.

THEY ADDED BARS TO WINDOWS…

A year later, the windows are all locked.

But they need more help! Auto-installed malware has infected all the smart refrigerators, which have Linux-based controllers.

But now you have a problem: disabling updates seems risky too!

PUZZLE: LINUX IS THE HOUSE…

We can do a reasonable job of securing Linux and its tools

But how can we secure the whole enchilada? Linux plus tools plus random applications?

Most enterprises have specific configurations or packages they approve, and you just aren't allowed to download others!

IN MODERN SYSTEMS, UPDATES AND APPLICATIONS ARE INCREASINGLY THE ISSUE!

Suppose Linux and your application were "perfect". Totally secure. You still would need to install helper applications and give them permission to accept and send requests.

Many employ components from open-source suppliers that don't necessarily use the best practices. Some could have been compromised long after they were first released.

If an application is insecure, it won't matter if Linux itself is secure: anything that application can read or update can be compromised.

KEN THOMPSON: HOW TO HIDE AN ATTACK

Suppose you want to attack the login program so that it always allows “Daffy Duck” to log in with no password.

You first modify the code for login to do this. But then modify the C compiler to pattern match for the original login code and to substitute this Daffy Duck thing *at compile time*.

Then you do the same trick to the compiler itself!

SOME PROTECTIONS ARE BUILT IN

For example, you can tell C++ to compile with address space randomization automatically performed.

You can also take compiler warnings seriously and can even use "proof tools" for ultra-sensitive portions of your code, like the algorithms used to decide which data to trust during self-repair

None of this will compensate for bugs… and you can't avoid bugs!

EXAMPLE: LET'S REVISIT THE ISSUE OF BOTS THAT TRY TO DISRUPT A DATA CENTER

TCP SYN Attack (DDoS) protection is important. This is a common attack on Linux servers in big datacenter settings, like Amazon

In these attacks, bots initiate connections but don't complete the 3-way handshake. This leaves a "pending connection" object in the server. Eventually the server runs out of memory and crashes.

PROTECTION AGAINST THESE ATTACKS?

To protect against a SYN attack, Linux dynamically slows the rate at which new TCP connections can be made.

The usual policy is an exponentially increasing delay: the first connection is accepted instantly, but the second only after a delay of 1ms, the next after 4ms, etc.

Delay grows as 2^k after k connection attempts.

CONSEQUENCE

On a server that isn't under attack, connections are very fast.

But if a server is attacked by bots, it only allows a smaller number of connections per second (the bot gets a timeout and must retry).

Harder to make a connection, but once you succeed, the server itself won't be ground to a halt by bot activity

UNDESIRED CONSEQUENCES?

It definitely is slower to make a connection. Moreover, some systems need a lot of TCP connections, and Linux forces them to occur slowly.

This is leading to a split between a style of system used in settings where we want SYN-attack protections and systems used inside data centers that want super-fast connection logic.

It forces a greater level of sophistication on the developers.

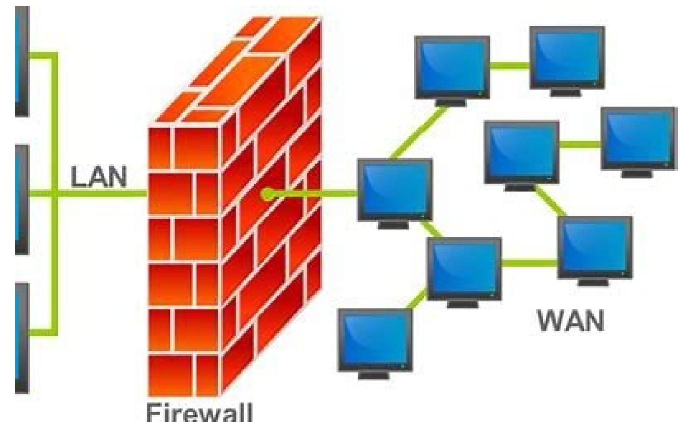
DDOS VIA REPLAY

Blocked from doing a TCP SYN attack, the attacker could just "tape record" network traffic for a few days and then replay the same packets at very high rates.

These will be ignored by TCP (they are old duplicates)

… but are not likely to be blocked by the firewall. It let them in the first time!

FIREWALLS

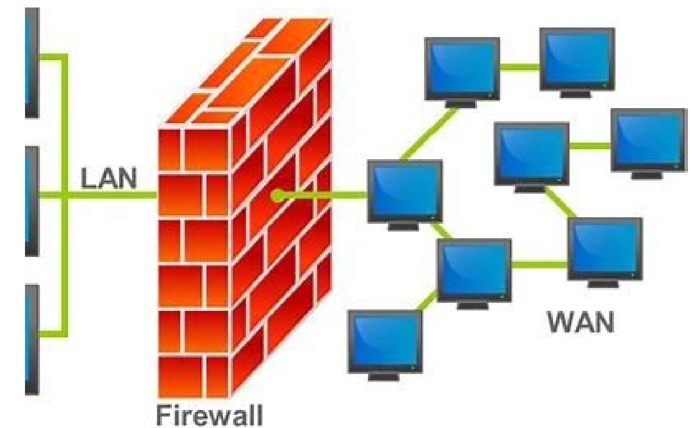


Firewalls are a powerful feature for protection.

Early firewalls simply blocked ports that aren't legitimately in use, but modern ones also have the ability to scan packets for payloads that match problematic signatures.

Hackers have fought back by designing attacks designed to look as legitimate as possible. This makes them harder to block.

THEY COME IN LAYERS



In a typical home or workplace, the Internet arrives at some form of "ingress box".

- This will be a powerful firewall that may even be able to examine packet contents at full line rates
- It will also do network address translation (NAT)
- It won't even expose computer names from inside the network unless the application explicitly publishes them via DNS.

This first barrier will stop many attacks

YOUR LINUX MACHINE *ALSO* HAS A FIREWALL

Different vendors have different names for this component. It can configure Linux as a router (!) and also is a firewall.

In Ubuntu, the "iptable" command controls the internal router and firewall capability.

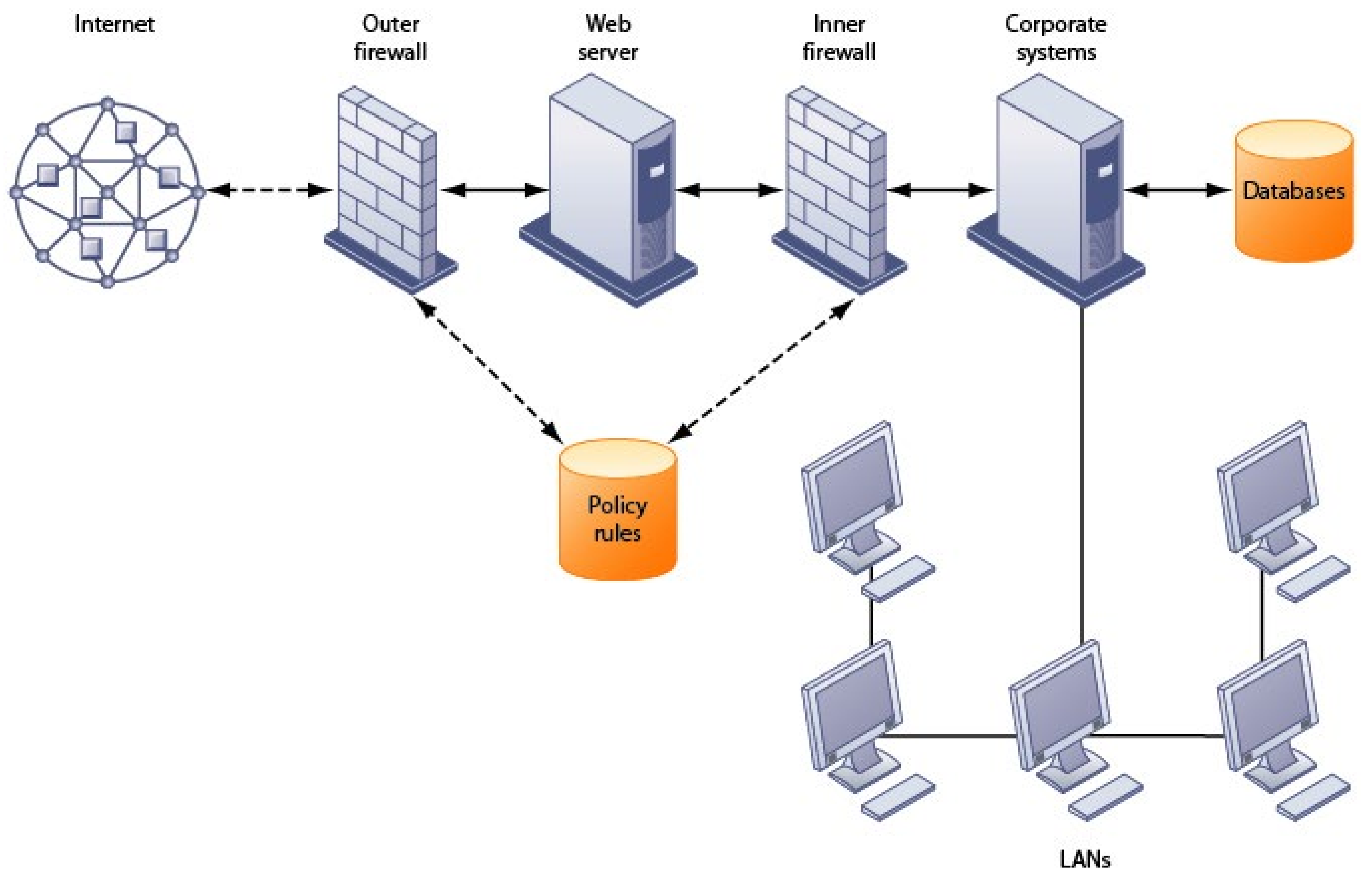
Controlled by "firewall rules" that you can configure/override.

EXAMPLES OF RULES

My MemCacheD servers are allowed to talk to one-another on port 9543, but only within IP domain 192.68.41.xxx

Block all incoming email connections to this machine.

Allow routing from subnet A to subnet B.


Internet
Outer firewall
Web server
Inner firewall
Corporate systems
Databases
Policy rules
LANs

WHAT IF SOMEHOW A VIRUS SLIPS IN?

The next stage of defense is concerned with limiting damage and discovering the virus to clean it up.

A big barrier is the Linux concept of user id’s and “group” ids (like a project team).

Each file has separate permissions for user, group and world.

HOW VIRUSES “SUBVERT” THE RULES

Some viruses try to trick the Linux system into giving the process they infect superuser privileges.

One old but still common trick: take over a console and display a mimic of the login screen. Save anything they type.

If someone does try to log in, print “User name / password combination unknown” and let the normal login run.

A VIRUS MIGHT ALSO TRY AND TRICK SOME PROGRAM WITH PRIVILEGES INTO "HELPING"

We saw this with the viruses that put their own files in special places.

The idea is to pick some task the elevated privilege programs do periodically and try and subvert that normal behavior to actually run the virus script with superuser permissions.

VIRUS SCANNERS

Most worms and viruses and bot-kits have recognizable “signatures”.

Companies have created honeypot systems just to see how attacks work and how infected systems “look”. From this they can construct patterns to recognize those signatures.

This enables them to scan both periodically and even block attacks in real-time by intercepting the incoming bootstrap logic.

BIG IDEA?

Instead of one firewall policy that is cast in stone, the policy can dynamically be configured.

In effect, understand how this virus attacks, then craft an anti-viral solution that watches for a “signature” of the attack and disrupts key steps

WHAT'S IN A SIGNATURE?

In fact these are really scripts.

"Look for files named … in folder …, quarantine them."

"Check the binary of program /bin/…, see if it has changed"

Etc.

FEATURES LIKE SYMBOLIC LINKS, DLL INTERPOSITION CAN BE MISUSED!

Linux symbolic links are files that "redirect" to some other file. We use them as a convenience, but a virus might exploit them!

DLL interposition is useful for extending or debugging a program, but a virus might try to use them to hijack your code.

/dev/proc is used for debugging. A virus might try to misuse it to see a remote login and password in memory

… THESE ARE HARD FOR VIRUS SCANNERS!

If a virus scanner blocks legitimate Linux functionality, many applications will break.

Yet many of these features are rarely used in real applications.

MILITARY-GRADE SOLUTIONS?

Some military systems are preconfigured in a menu of specific versions.

The user is authorized to use a specific system configuration.

The virus scanner simply checks that the system is *exactly* the same as the original menu option, except for application data

CLEAN ROOM CODING APPROACHES

Companies adopt coding standards: Not just "use C++" but "document your code this way." "Solve this kind of problem using this specific library".

Code is carefully specified, designed, reviewed.

Every element is subject to compliance testing and acceptance testing. Many eyes on each line.

TYPE CHECKING HELPS A LOT!

Type checking is never the whole story. But stronger checking reduces the rate of bugs and flaws by orders of magnitude.

In the limit (languages like Daffny, Rust) “types” can even include assertions, proofs, invariants. At Cornell we are big fans of this!

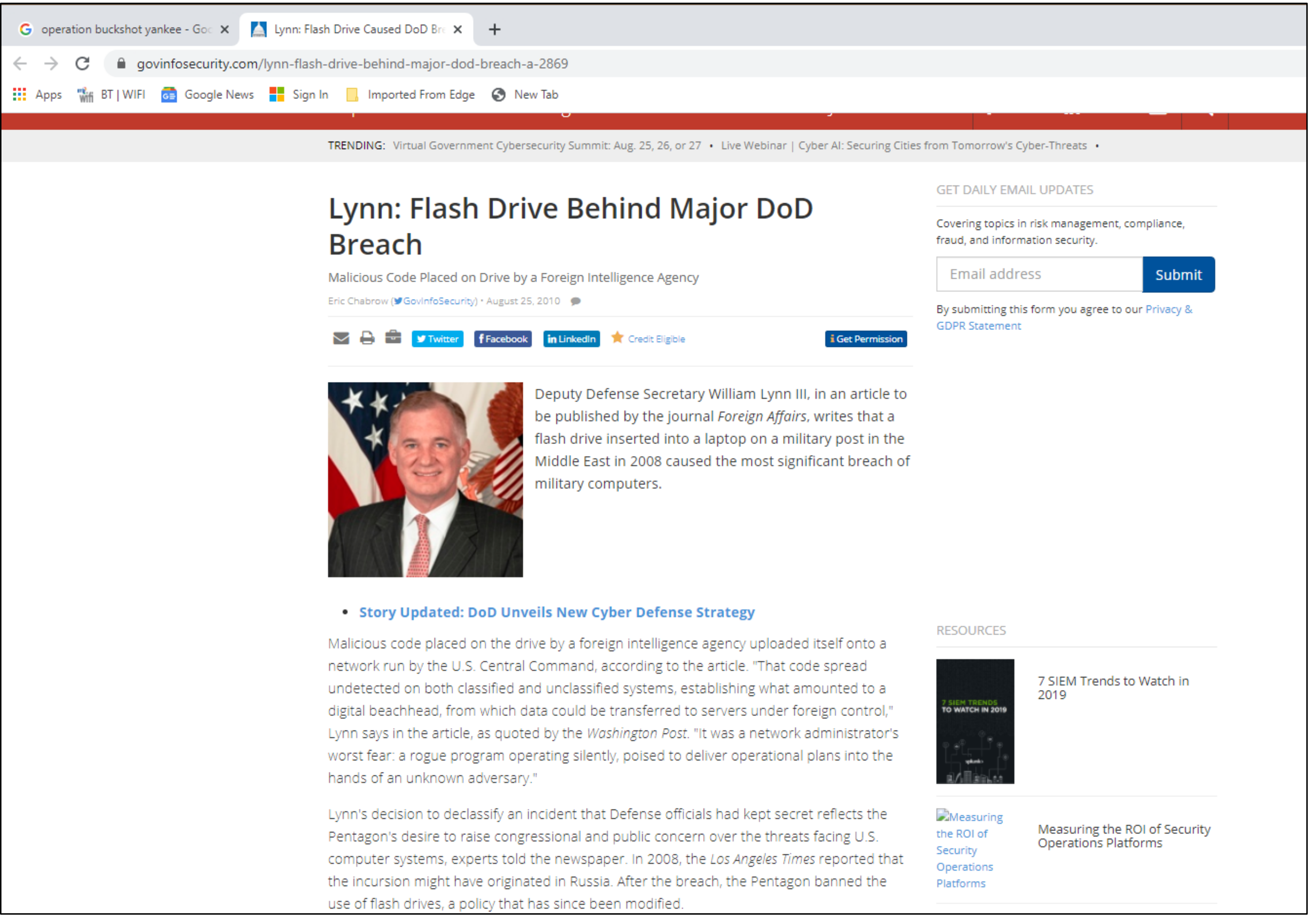
Techniques like these lead to hardened, much safer solutions!

CAN SYSTEMS REALLY BE PROTECTED?

Recall that article from Lecture 7!

Intruders left really appealing "new" USB drives with huge capacity in places like a men's room shelf.

Foolishly, others saw these and took them and plugged them in. Hidden virus software was able to break into their machines!



operation buckshot yankee - Goo | Lynn: Flash Drive Caused DoD Br

govinfosecurity.com/lynn-flash-drive-behind-major-dod-breach-a-2869

Apps | BT | WIFI | Google News | Sign In | Imported From Edge | New Tab

TRENDING: Virtual Government Cybersecurity Summit: Aug. 25, 26, or 27 • Live Webinar | Cyber AI: Securing Cities from Tomorrow's Cyber-Threats •

Lynn: Flash Drive Behind Major DoD Breach

Malicious Code Placed on Drive by a Foreign Intelligence Agency

Eric Chabrow (GovInfoSecurity) • August 25, 2010

Twitter | Facebook | LinkedIn | Credit Eligible | Get Permission

Deputy Defense Secretary William Lynn III, in an article to be published by the journal *Foreign Affairs*, writes that a flash drive inserted into a laptop on a military post in the Middle East in 2008 caused the most significant breach of military computers.

- **Story Updated: DoD Unveils New Cyber Defense Strategy**

Malicious code placed on the drive by a foreign intelligence agency uploaded itself onto a network run by the U.S. Central Command, according to the article. "That code spread undetected on both classified and unclassified systems, establishing what amounted to a digital beachhead, from which data could be transferred to servers under foreign control," Lynn says in the article, as quoted by the *Washington Post*. "It was a network administrator's worst fear: a rogue program operating silently, poised to deliver operational plans into the hands of an unknown adversary."

Lynn's decision to declassify an incident that Defense officials had kept secret reflects the Pentagon's desire to raise congressional and public concern over the threats facing U.S. computer systems, experts told the newspaper. In 2008, the *Los Angeles Times* reported that the incursion might have originated in Russia. After the breach, the Pentagon banned the use of flash drives, a policy that has since been modified.

GET DAILY EMAIL UPDATES

Covering topics in risk management, compliance, fraud, and information security.

Email address | Submit

By submitting this form you agree to our Privacy & GDPR Statement

RESOURCES

7 SIEM Trends to Watch in 2019

Measuring the ROI of Security Operations Platforms

THE CORE PROBLEM IS A MIX OF COMPLEXITY AND HUMAN ERROR

The platforms we use are huge and complex and even the hardware is quite hard to configure properly.

The resulting code is much harder to verify than code to build a B+ tree or sort a list. We can only harden some parts.

Meanwhile, humans have limitations, and make mistakes

VIRTUALIZATION ATTACKS ARE TOUGHEST

In these attacks, the virus controls the hardware, but then creates a virtual environment that looks identical to the hardware.

User code and virus scanners run inside Linux… in the virtual environment. They just won't see the virus… they can't!

The virus is in control, yet totally invisible.

EXAMPLE: INFORMATION FLOW REFERENCE MONITORS

Idea here is to abstractly model applications and data

Design a flow graph that represents permitted and non-permitted data flows. For example, a smart home might be permitted to use cameras and microphones yet only allowed "share" anonymous summary data of energy use.

Then build a monitor to enforce these restrictions.

VIRTUALIZED NETWORK & SYSTEM

Aggressive virtualization (installed by the attacker) is probably the hardest thing to protect against.

The watcher components won't realize they aren't seeing the true system, or the true network. So they don't trigger even though an exploit is actively occurring!

Kind of like the Matrix: Inside the matrix you don't see the truth

VIRTUALIZED NETWORK & SYSTEM

Professor Weatherspoon and his students had programmable high-speed NICs for a modern network.

Original idea: use the NIC to monitor network traffic.

Actual outcome? A bit more “ambiguous”

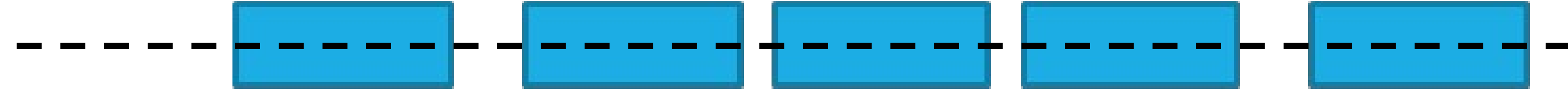
VIRTUALIZED NETWORK & SYSTEM

Professor Weatherspoon and his students had programmable high-speed NICs for a modern network.

He showed that he could virtualize the network itself. His NICs are able to subvert most forms of monitoring.

Issue? The network monitor doesn't see the deepest level of the network itself!

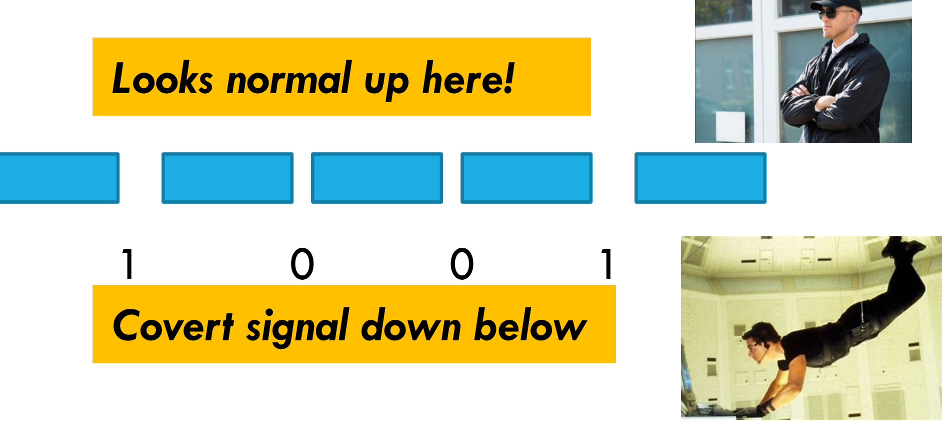
Looks normal up here!



1 0 0 1

Covert signal down below

UNDER THE SURFACE



In fact, Hakim's programmable NICs were encoding information into the spacing between packets.

For example, if the "space" was of length 0.5us, this is a 0 bit. If the space has length 1us, this is a 1 bit. Monitors can't see this spacing: only the NIC itself had access to this form of information.

Modern networks have continuous "no-op" traffic.... Lots of packets.

SUPPOSE THE NETWORK CAN SEND 75M PACKETS PER SECOND ON EACH LINK

This is about 10MB/second, per link.

As fast as an internet into a normal home!

His network could quietly copy data day and night for months and even a high-quality network monitor wouldn't see a thing!

CAN YOU PROTECT AGAINST THIS?

Easily, if you know this is happening.

A store and forward router (that uses NICs lacking programmable functionality!) can randomize the spacing.

So Hakim’s network-on-a-network is an example of a subtle hack, yet one you could easily defend against if you knew!

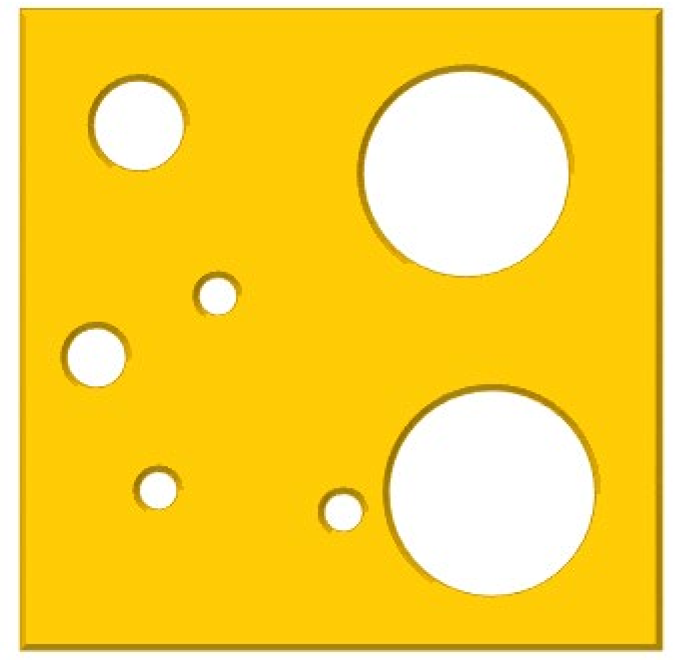
BROADER REMEDY? DEFENSE "IN DEPTH"

Many sensitive systems, like hospitals, are isolated behind multiple levels of firewalls, gateways and "air gaps".

They may use multiple forms of protection internally, too.

And there are sometimes even "honeypot" traps designed to lure intruders as a way of tricking them into revealing themselves

DEFENSE IN DEPTH VISUALIZED AS "SWISS CHEESE" (FROM A NYT COVID ARTICLE)



https://tomaspueyo.medium.com/coronavirus-the-swiss-cheese-strategy-d6332b5939de

A MULTI-LAYERED DEFENSE REALLY WORKS!

Each layer uses different techniques

Some might be like firewalls, others like reference monitors, others could do things like address space randomization. Throw in some honeypot systems and monitor them continuously!

Add them together… and a hacker has a very high risk of being caught in the act.

IT'S A JUNGLE OUT THERE!

Linux and C++ seem pretty innocent

Yet serious systems run in a very hostile world!

Using the tools carefully is the best defense. Build every program as if it might be used for decades!

GOOD LUCK!

… on prelim2, and on the final if you opt to take it

… and good luck in your career! We hope that C++ will be a really valuable tool and that all the stuff you learned in CS4414 will bootstrap into a great start at a great job!