Operating Systems Security

PROFESSIONAL EDUCATION

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

© 2015-2016 Massachusetts Institute of Technology

Operating Systems Security

Frans Kaashoek

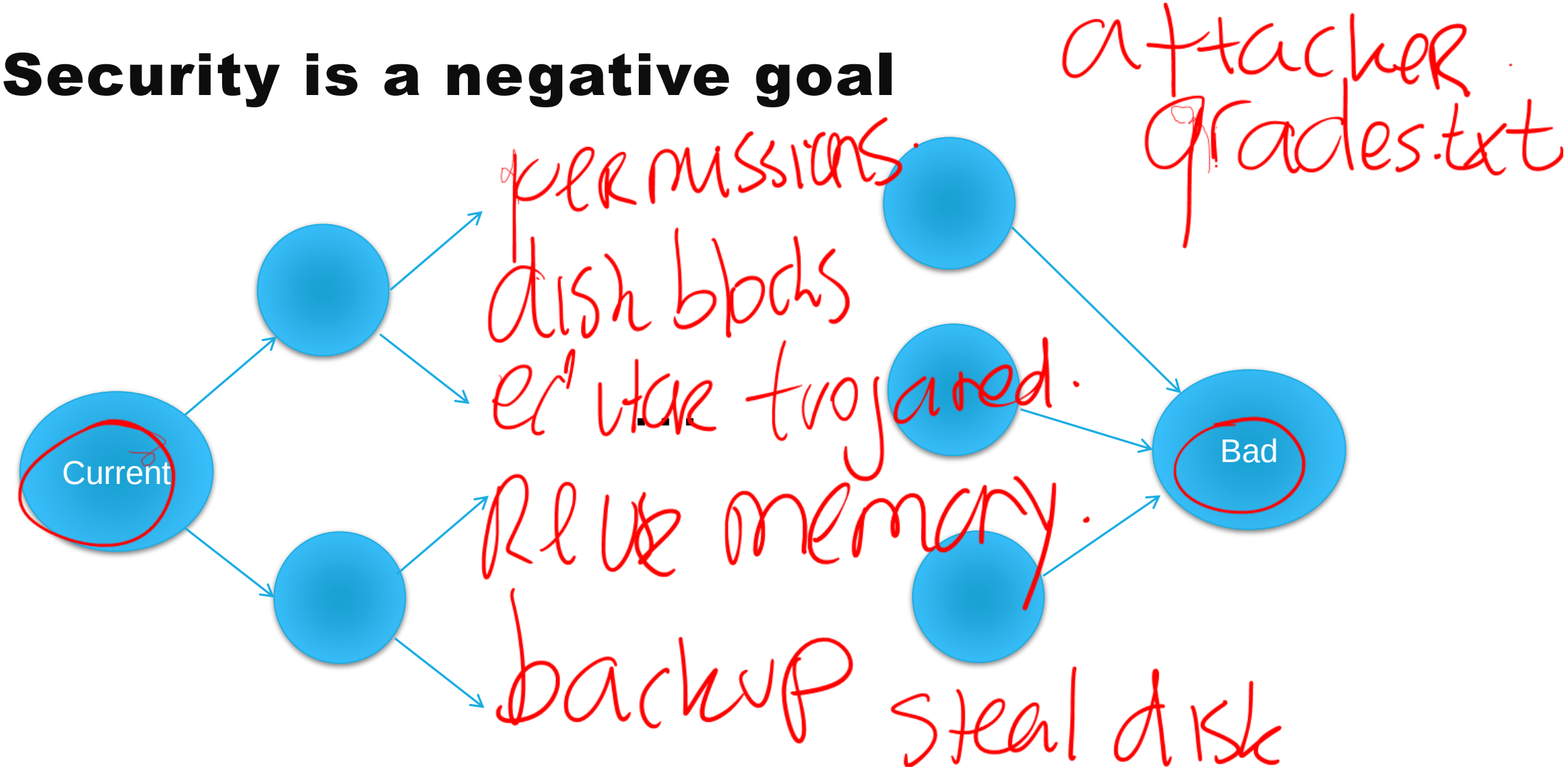
The Charles Piper Professor of Computer Science

Computer Science and Artificial Intelligence Laboratory (CSAIL)

Massachusetts Institute of Technology

© 2015-2016 Massachusetts Institute of Technology

Security is a negative goal



Many ways for an attacker to break system

© 2015-2016 Massachusetts Institute of Technology

Difficult to achieve negative goals

The designer must protect against *all* possible ways that attacker can break system

The designer is likely to miss one path

How to mitigate problems?

© 2015-2016 Massachusetts Institute of Technology

This session: three big ideas

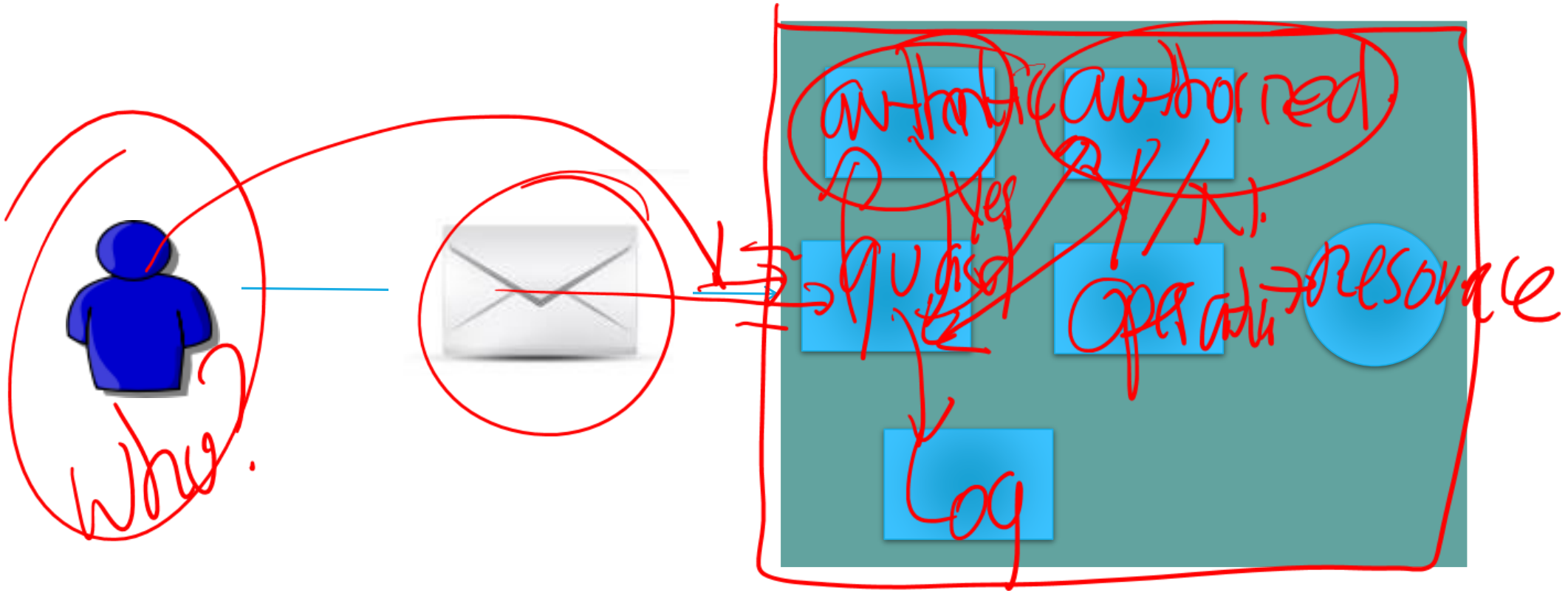
1. Complete mediation
2. Privilege separation
3. Minimize trusted computing base

This is difficult; bugs are in unexpected places

Case study: undefined behavior bugs

© 2015-2016 Massachusetts Institute of Technology

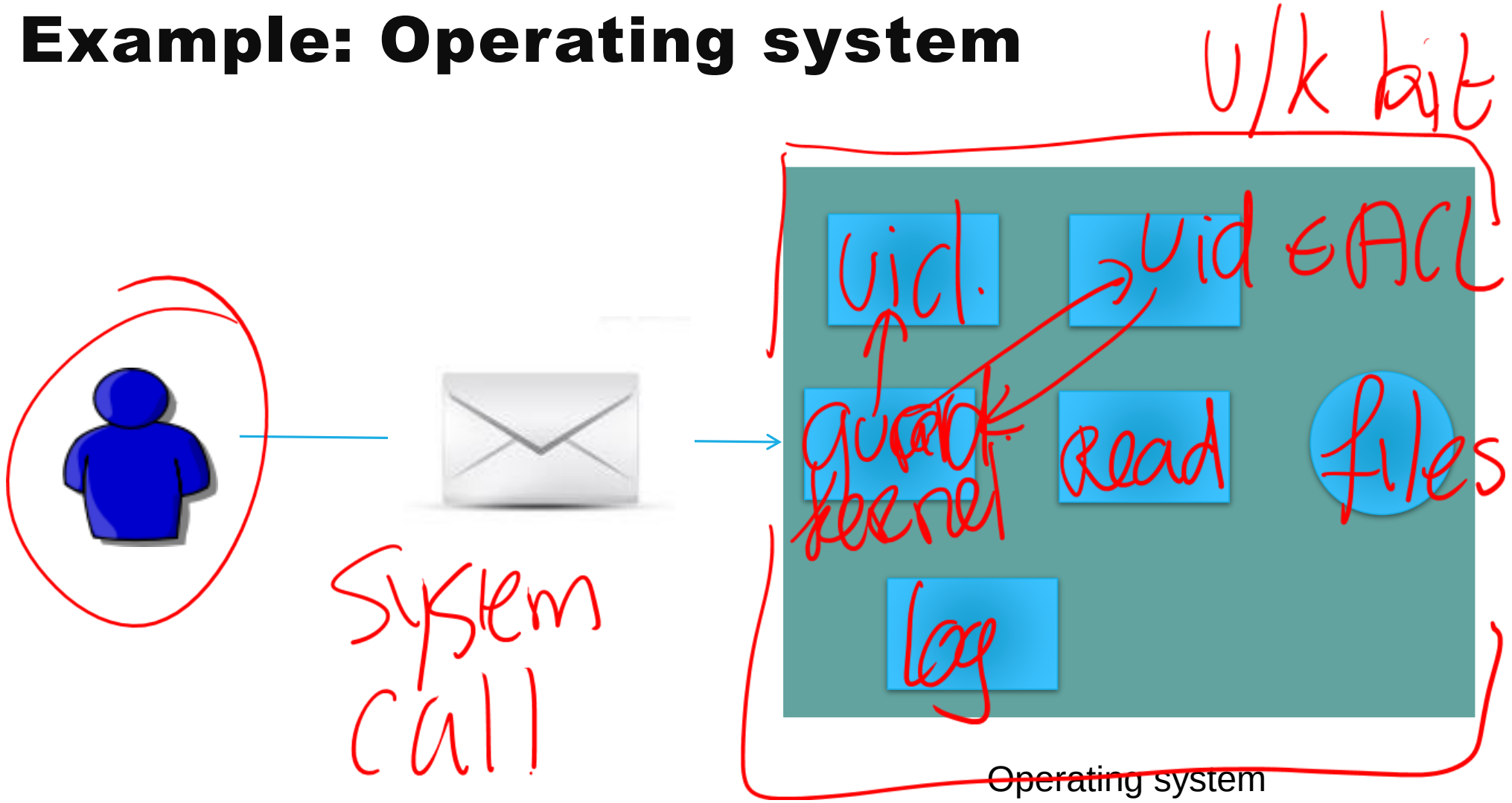
Complete mediation w. guard model



Computer system

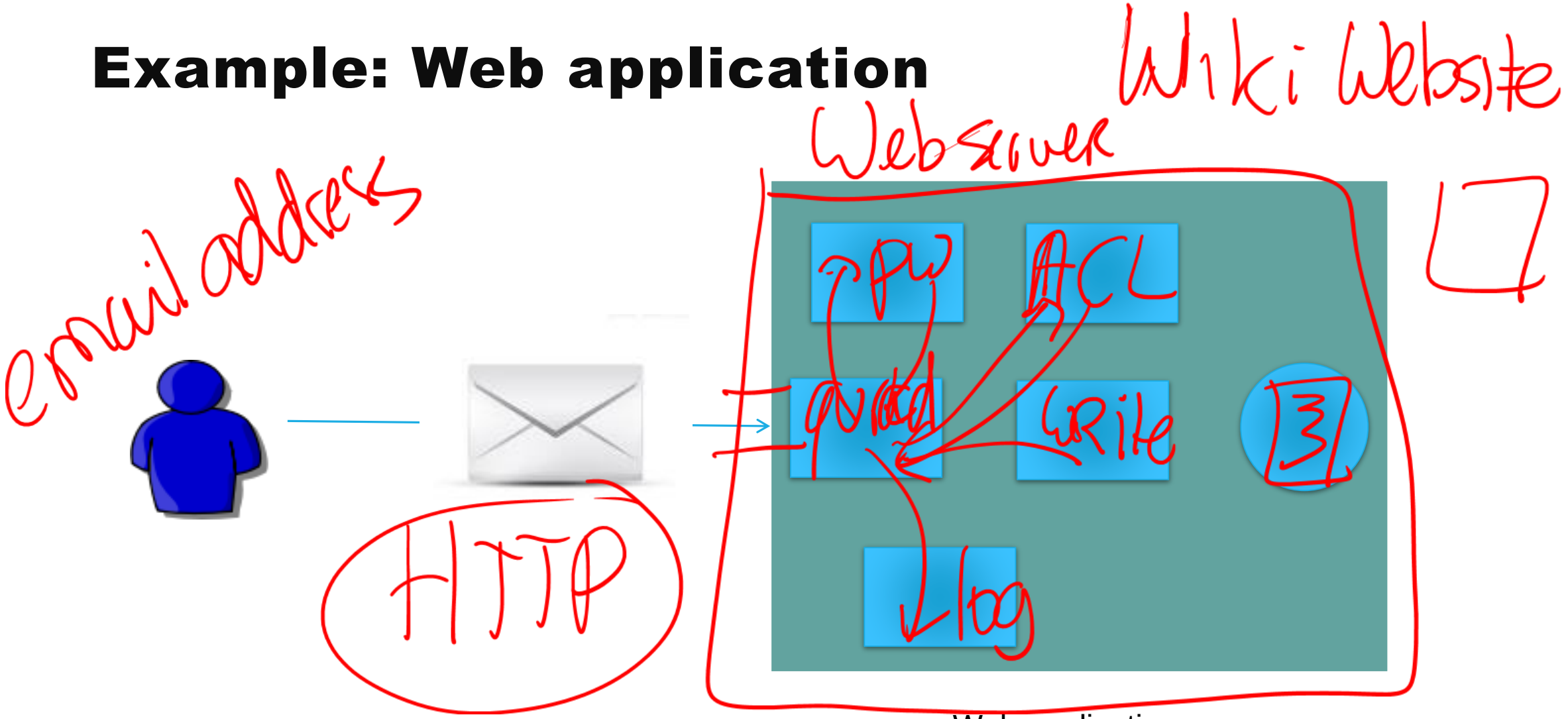
© 2015-2016 Massachusetts Institute of Technology

Example: Operating system



© 2015-2016 Massachusetts Institute of Technology

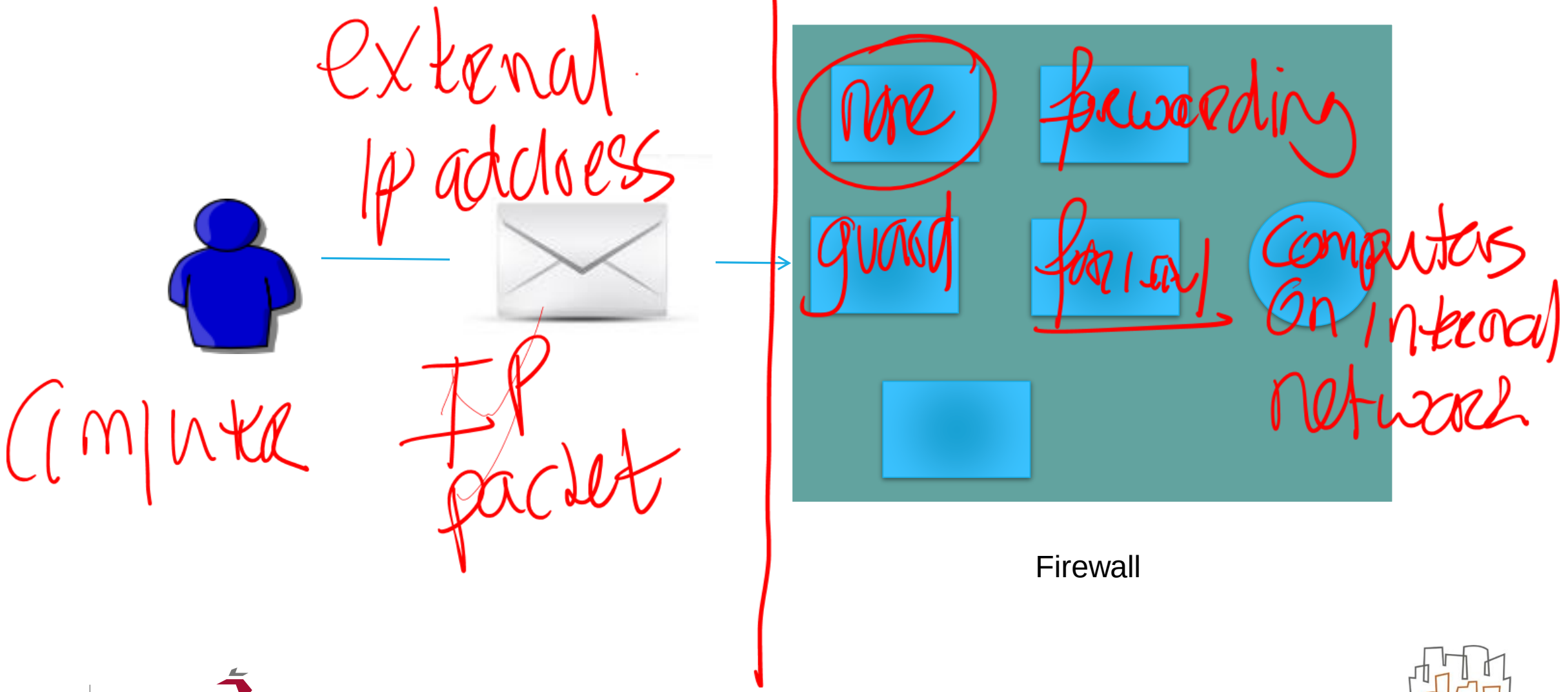
Example: Web application



Web application

© 2015-2016 Massachusetts Institute of Technology

Example: firewall



Firewall

© 2015-2016 Massachusetts Institute of Technology

Model is simple. What can go wrong?

Complete mediation is challenging

- All ways to access must be checked; backdoors?

Software bugs in mediation

- One bug per ~1,000 lines of code

Mismatch between policy and mechanism

- What should the firewall rules be?

Policy is challenging to get right

- Example: security questions are easier to guess than password
- No way to enforce “Only if user forgets password, then …”

© 2015-2016 Massachusetts Institute of Technology

Summary so far

Guard model is good, but hard to get right

Any principled ideas to improve further?

© 2015-2016 Massachusetts Institute of Technology

Separation of privilege

Split system into modules and give each module the *least* privilege to do its job

Examples:

1. Use several physical machines: run database on different machine than web site
2. Use virtual machines to split
3. Split applications in components

© 2015-2016 Massachusetts Institute of Technology

Example: bad design shopping web site

Design:

- Requests: product searches, take orders
- Store product info and orders in same DB

Bad properties:

- Bug in search software (e.g., SQL injection) can expose credit card numbers
- *All* software is *trusted*

© 2015-2016 Massachusetts Institute of Technology

Example: better design shopping web site

Idea: reduce trusted code

Two servers:

1. One server doing search
2. One server doing orders

Good properties:

- If no interaction between servers, search cannot obtain credit card numbers
- Search software is now *untrusted*.

© 2015-2016 Massachusetts Institute of Technology

Challenges in privilege separation

1. Modules need to share
   Search software may need access to orders database
2. Need system support
   Must be able to compartmentalize servers
   Support controlled sharing
3. Need to configure privileges carefully
   Give servers privileges for accessing only their DB
4. Performance
   Two servers maybe less efficient than 1
5. Reduce trusted software

© 2015-2016 Massachusetts Institute of Technology

Trust computing base (TCB)

All software that must be trusted to achieve security

Big goal in security: *reduce* TCBs

less software ➔fewer bugs ➔fewer exploits

© 2015-2016 Massachusetts Institute of Technology

Challenge: bug-free TCB

Bug in TCB undermines privilege separation

TCB are often large and complex (e.g., kernel)

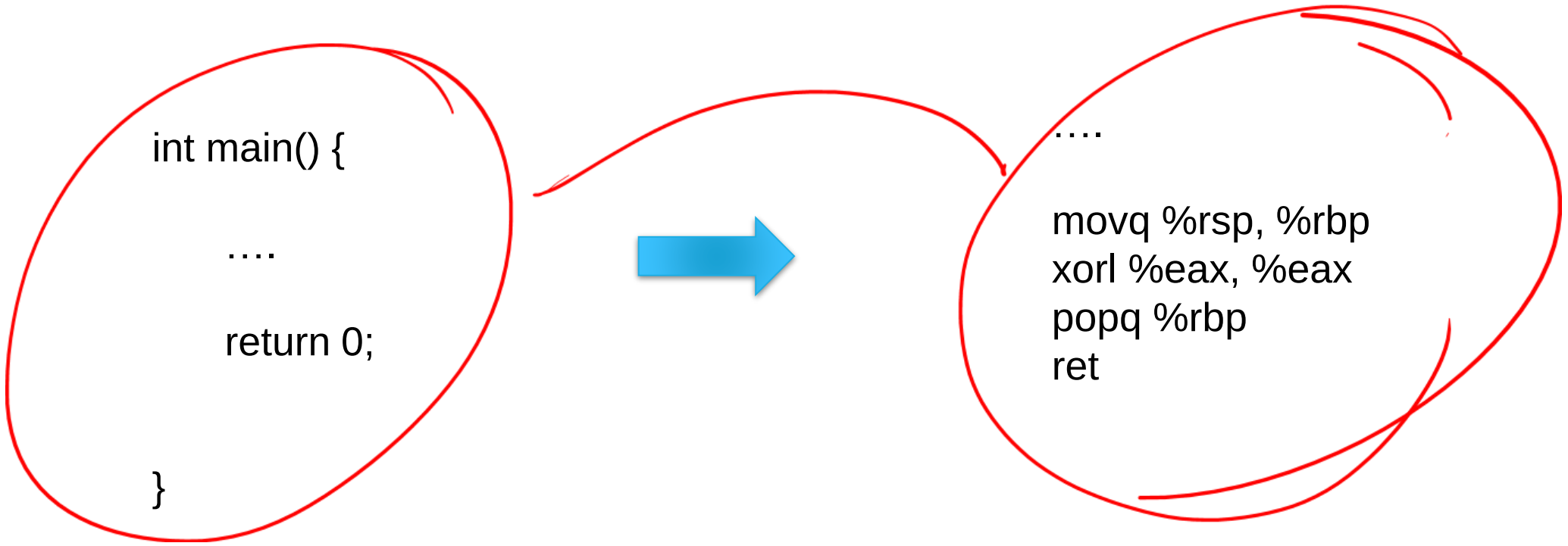
- Large *and* complex ➔ bugs
- Many types of bugs

New classes of bugs are discovered

- Case study: undefined behavior [SOSP 2013]

© 2015-2016 Massachusetts Institute of Technology

Belief: compiler is faithful translator



Not true if code invokes undefined behavior

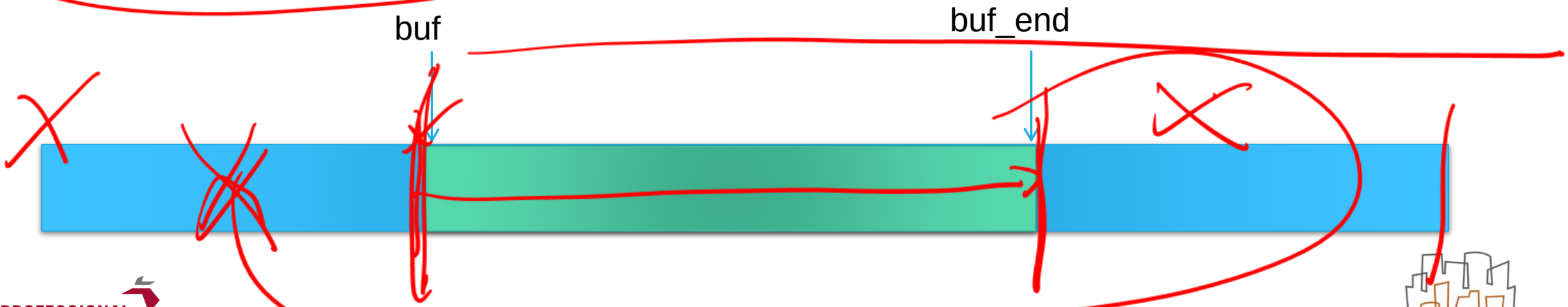
This has bad security implications

© 2015-2016 Massachusetts Institute of Technology

Example: compiler discards sanity check

```
char *buf = …;
char *buf_end = …;
unsigned int off = /* supplied by untrusted source */
if (buf + off >= buf_end)
    return;
if (buf + off < buf)
    return;
/* access buf[0..off-1] */
```

Linux kernel
Chrome
Python interpreter



© 2015-2016 Massachusetts Institute of Technology

Undefined behavior allows such optimziations

Undefined behavior: the spec "imposes no requirements"

E.g., pointer overflow (buf+off)

Original goal: emit efficient code

E.g., no bounds checks emitted. Good on hardware that does bounce checking in hardware.

Now compilers optimize code with undefined behavior away.

© 2015-2016 Massachusetts Institute of Technology

Examples of undefined behavior in C

Meaningless checks from real code:

Pointer overflow:	if (p + 100 < p)
Signed integer overflow:	if (x + 100 < x)
Oversized shift:	if (!(1 << x))
Null pointer dereference:	*p; if (p)
Absolute value overflow:	if (abs(x) < 0)

© 2015-2016 Massachusetts Institute of Technology

Unstable code confuses programmers

Unstable code: compilers discard code due to undefined behavior

Compiler writers assume programmer *never* invokes undefined behavior. If there is undefined behavior, programmer could not have intended it. So, it is ok to optimize it away.

Unfortunately, programmers don't know the exact semantics of C, and read write code with undefined behavior without realizing it.

Effects:

- Security checks discarded
- Weakness amplified
- Unpredictable system behavior

© 2015-2016 Massachusetts Institute of Technology

Unstable code confuses programmers

"This optimization will create MAJOR SECURITY ISSUES in ALL MANNER OF CODE. I don't care if your language lawyers tell you gcc is right …. FIX THIS! NOW!"

A gcc user

Bug #30475 assert(int + 100> int) optimized away

© 2015-2016 Massachusetts Institute of Technology

Response from gcc developers

"I am sorry that you wrote broken code to begin with … GCC is not going to change."

© 2015-2016 Massachusetts Institute of Technology

Test existing compilers

12 C/C++ compilers

- gcc
- aCC (HP)
- icc (Intel)
- open64 (AMD)
- suncc (Oracle)
- ti (TMS320C6000)
- clang
- armcc (ARM)
- msvc (Microsoft)
- pathcc (PathScale)
- xlc (IBM)
- windriver (Diab)

© 2015-2016 Massachusetts Institute of Technology

Compilers often discard unstable code

	`if` $(p + 100 < p)$	$*p$; `if` $(!p)$	`if` $(x + 100 < x)$	`if` $(x^+ + 100 < 0)$	`if` $(!(1 << x))$	`if` $(\text{abs}(x) < 0)$
gcc-2.95.3	–	–	O1	–	–	–
gcc-3.4.6	–	O2	O1	–	–	–
gcc-4.2.1	O0	–	O2	–	–	O2
gcc-4.8.1	O2	O2	O2	O2	–	O2
clang-1.0	O1	–	–	–	–	–
clang-3.3	O1	–	O1	–	O1	–
aCC-6.25	–	–	–	–	–	O3
armcc-5.02	–	–	O2	–	–	–
icc-14.0.0	–	O2	O1	O2	–	–
msvc-11.0	–	O1	–	–	–	–
open64-4.5.2	O1	–	O2	–	–	O2
pathcc-1.0.0	O1	–	O2	–	–	O2
suncc-5.12	–	O3	–	–	–	–
ti-7.4.2	O0	–	O0	O2	–	–
windriver-5.9.2	–	–	O0	–	–	–
xlc-12.1	O3	–	–	–	–	–

© 2015-2016 Massachusetts Institute of Technology

Observations

Compilers silently remove unstable code

Compilers have become more aggressive over time

Different compilers behave differently

Change/upgrade compiler ➔ broken system

➔Need a systematic approach

© 2015-2016 Massachusetts Institute of Technology

Our approach: precisely flag unstable code

C/C++ source ➔ LLVM IR ➔

STACK ➔ warnings

```
% ./configure
% stack-build make    # intercept cc
% poptck                    # run STACK
```


© 2015-2016 Massachusetts Institute of Technology

Design overview of STACK

What's the difference, compilers vs most programmers?

- Assumption Δ: programs don't invoke undefined behavior

What can compilers do only with assumption Δ?

- Optimize away unstable code

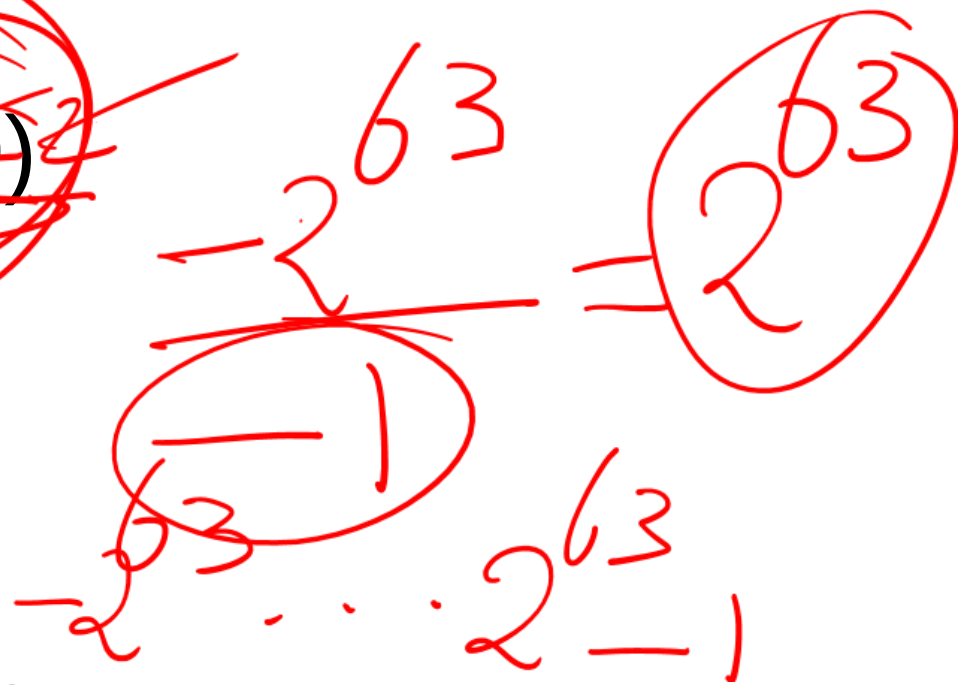
Stack: mimic a compiler that selectively enables Δ

- Phase 1: optimize without Δ
- Phase 2: optimize with Δ
- Unstable code: differences between the two phases

© 2015-2016 Massachusetts Institute of Technology

Example of identifying unstable code

1. res = x/y;
2. if (y == -1 && x < 0 && res < 0)
3. return; /* $-2^{63}/-1 < 0$ */



Assumption:

No division by zero: y ≠ 0

No division overflow: y ≠ -1 or x ≠ INT_MIN

STACK can optimize “res < 0” to false only with Δ

Phase 1: is res < 0 = false in general? No

Phase 2: is res < 0 = false with Δ? Yes.

Report res < 0 as unstable code.

© 2015-2016 Massachusetts Institute of Technology

STACK finds new bugs

Applied STACK to many popular systems

Inspected warnings and submitted patches to developers

Binutils, Bionic, Dune, e2fsprogs, Ffmpeg+Libav, file, FreeType, GMP, GRUB, HiStar, Kerberos, libX11, libarchive, libgcrypt, Linux kernel, Mosh, Mozilla, OpenAFS, OpenSSH, OpenSSL, PHP, plan9port, Postgres, Python, QEMU, Ruby+Rubinius, Sane, uClibc, VLC, Wireshark, Xen, Xpdf

Developers accepted most of our patches

160 new bugs

© 2015-2016 Massachusetts Institute of Technology

STACK warnings are precise

Kerberos: STACK produced 11 warnings

Developers accepted every patch
No warnings for fixed code
Low false warning rate: 0/11

Postgres: STACK produced 68 warnings

9 patches accepted: server crash
29 patches in discussion: developers blame compilers
26 time bombs: can be optimized away by future compilers
4 false warnings: benign redundant code
Low false warning rate: 4/68

© 2015-2016 Massachusetts Institute of Technology

Unstable code is prevalent

Applied STACK to all Debian Wheezy packages

- 8,575 C/C++ packages
- ~150 days of CPU time to build and analyze

STACK warns in ~40% of C/C++ packages

© 2015-2016 Massachusetts Institute of Technology

How to avoid unstable code

Programmers

Fix bugs

Workaround: disable certain optimizations

Compilers & checkers

Many bug-finding tools fail to model C spec correctly

Use our ideas to generate better warnings

Language designers: revise the spec

Eliminate undefined behavior? Perf impact?

© 2015-2016 Massachusetts Institute of Technology

Stepping back: how to avoid bugs

Bugs are the source of many security vulnerabilities

Compartmentalize bugs using privilege separation

Minimize TCB

Next session will talk about approaches to eliminate certain kinds of bugs

© 2015-2016 Massachusetts Institute of Technology

THANK YOU

Frans Kaashoek

The Charles Piper Professor of Computer Science

© 2015-2016 Massachusetts Institute of Technology