rust / sandboxing 1

last time

static analysis possibilities

- very practical analyses: finding common mistake patterns
- more ambitious/inaccurate: safety properties like no UAF/out-of-bounds

information flow analysis / static or dynamic

- does data move from source to sink
- e.g. does private info (keypresses, phone numbers, etc.) leak?
- e.g. can attacker control SQL statement, shell command?
- e.g. where does malware's network input end up on filesystem?

toward better programming languages

- what about C/C++ makes people use it
- idea of escape hatches for limits of safe languages

logistics note

FUZZ deadline

quiz

simple Rust syntax (1)

```
fn main() {
    println!("Hello, World!\n");
}
```

simple Rust syntax (2)

```
fn timesTwo(number: i32) -> i32 {
    return number * 2;
}
```

simple Rust syntax (3)

```
struct Student {
    name: String,
    id: i32,
}

fn get_example_student() -> Student {
    return Student {
        name: String::from("Example Fakelastname"),
        id: 42,
    };
}
```

simple Rust syntax (4)

```rust
fn factorial(number: i32) -> i32 {
    let mut result = 1;
    let mut index = 1;
    while index <= number {
        result *= index;
        index = index + 1;
    }
    return result;
}
```

simple Rust syntax (4)

```
fn factorial(number: i32) -> i32 {
    let mut result = 1;
    let mut index = 1;
    while inde
        result
        index
    }
    return result;
}
```

“input” is a mutable variable

type automatically inferred as i32 (32-bit int)

Rust references

```
fn main() {
    let mut x: u32 = 42;

    {
        let y: &mut u32 = &mut x;
        *y = 100;
    }

    let z: &u32 = &x;

    println!("x = {}; z = {}", x, x);
}
```

Rust example

```rust
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

Rust example

```
use std::io;

fn main() {
    println!("Ent
    
    let mut inp
    // could have
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

"input" is a mutable variable
type is automatically inferred as String

Rust example

```
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut inp

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

pass mutable reference to input

Rust example

```rust
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdi

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

number is an immutable unsigned 32-bit integer

rules to stop dangling pointers (1)

objects have an single owner

owner is the only one allowed to modify an object

owner can give away ownership

simplest version: only owner can access object

never have multiple references to object — always move/copy

Rust objects and ownership (1)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(vector);
    // **moves** vector into mysum()
        // philosophy: no implicit expensive copies

    println!("Sum is {}", sum);
    // ERROR
    println!("vector[0] is {}" , vector[0]);
}
```

Rust objects and ownership (1)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo
    le
    le
    //


    pr
    // ERROR
    println!("vector[0] is {}" , vector[0]);
}
```

```
   Compiling lecture-demo v0.1.0 (file:///home/cr4bd/spring2017/cs4630/...
error[E0382]: use of moved value: vector
  --> src/main.rs:16:34
   |
13 |     let sum = mysum(vector);
   |                     ------ value moved here
...
16 |     println!("vector[0] is {}" , vector[0]);
   |                                  ^^^^^^ value used here after move
```

Rust objects and ownership (2)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(vector.clone());
    // give away a copy of vector instead
        // mysum will dispose, since it owns it

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

Rust objects and ownership (2)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = ve
    let sum = mysum(vector.cl
    // give away a copy of vector instead
        // mysum will dispose, since it owns it

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

mysum borrows a copy

moving?

moving a Vec — really copying a pointer to an array and its size

cloning a Vec — making a copy of the array itself, too

Rust defaults to moving non-trivial types

some trivial types (u32, etc.) are copied by default

Rust objects and ownership (3)

```
fn mysum(vector: Vec<u32>) -> (u32, Vec<u32>) {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return (total, vector)
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let (sum, newVector) = mysum(vector);
    // give away vector, get it back

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

Rust objects and ownership (3)

```
fn mysum(vector: Vec<u32>) -> (u32, Vec<u32>) {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return (total, vector)
}

fn foo() {
    let vector: Vec
    let (sum, newVec
    // give away vec

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

mysum “borrows” vector, then gives it back
uses pointers

ownership rules

exactly one owner at a time

giving away ownership means you can't use object

either give object new owner or deallocate

ownership rules

exactly one owner at a time

giving away ownership means you can't use object

common idiom — temporarily give away object

either give object new owner or deallocate

ownership exercise

If called like `p = foo(p)`, which follow single-owner rule?

```
// (A)
char *foo(char *p) {
    free(p);
    return NULL;
}

// (B)
char *foo(char *p) {
    p = realloc(p, strlen(p) + 100);
    strcat(p, "test");
    return p;
}
```

```
// (C)
char *global;
char *foo(char *p) {
    if (p) free(p);
    return global;
}

// (D)
char *foo(char *p) {
    p[0] = 'A';
    return p;
}
```

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently

- object is "moved"

owner can let someone borrow object **temporarily**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

borrowing

```
fn mysum(vector: &Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(&vector);
    // automates (vector, sum) = mysum(vector) idea

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , vector[0]);
}
```

dangling pointers?

```c
int *dangling_pointer() {
    int array[3] = {1,2,3};
    return &array[0]; // not an error
}
```

```rust
fn dangling_pointer() -> &mut i32 {
    let array = vec![1,2,3];
    return &mut array[0]; // ERROR
}
```

dangling pointers?

```
int *dangling_pointer() {
    int array[3] = {1,2,3};
    return &array[0]; // not an error
}
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:19:25
   |
19 | fn dangling_pointer() -> &mut i32 {
   |                          ^ expected lifetime parameter
   |
   = help: this function's return type contains a borrowed value,
           but there is no value for it to be borrowed from
```

applying rules (1)

single owner, someone can borrow temporarily

only modify if exactly one user

```
let mut x = 42;     // (1)  int x = 42;         // (1)
let p = &mut x;     // (2)  int *p = &x;        // (2).
*p = 10;            // (3)  *p = 10;            // (3)'
println!("{}", x); // (4)  printf("%d\n", x); // (4)
```

Exercise 1/2/3/4: The owner of x on line 1/2/3/4 is:

A. (original owner) the variable x
B. (borrowed) the pointer/reference p

applying rules (2)

single owner, someone can borrow temporarily

only modify if exactly one user

```
let mut x = 42;     // (1)  int x = 42;          // (1)
let p = &mut x;     // (2)  int *p = &x;         // (2)
*p = 10;            // (3)  *p = 10;             // (3);
println!("{}", x); // (4)  printf("%d\n", x); // (4)
*p = 11;            // (5)  *p = 11;             // (5)
```

Rust rufuses to compile left-side: x being used while borrowed by p

Which changes would avoid this problem?

A. use $\star$p in the println!
B. make p mutable, reassign `p = &mut x` after line (4)
C. take a non-mutable reference to x instead of a mutable one

why lifetimes? (1)

```
let x = vec![1, 2, 3, 4];
let mut q = &x[1];
{
    let mut r = &x[1];
    let y = vec![5, 6, 7, 8];
    if random() == 0 {
        r = &y[1]; // SHOULD BE FINE
        q = &y[1]; // SHOULD BE ERROR
    }
    println!("{}", *r);
}
println!("{}", *q);
```

need to prevent q referring to values that live too long

why lifetimes? (2)

```
fn mystery(ptr: &i32, vec: &Vec<i32>) —> &i32 {...}

fn example() {
    ...
    let mut x = vec![1, 2, 3, 4];
    let mut q = &x[1];
    {
        let mut y = vec![5, 6, 7, 8];
        q = mystery(q, &y);
    }
    println!("{}", *q);
}
```

question: should assignment to be q from mystery be allowed?

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently

- object is “moved”

owner can let someone borrow object **temporarily**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

lifetimes

every reference in Rust has a lifetime

intuitively: how long reference is usable

Rust compiler infers and checks lifetimes

lifetime rules

object is borrowed for duration of reference lifetime

- can't modify object during lifetime
- can't let object go out of scope during lifetime

lifetime of function args must include whole function call

references returned from function must have lifetimes

- based on arguments or static (valid for entire program)

references stored in structs must have lifetime longer than struct

lifetime inference

```
fn get_first(values: &Vec<String>) -> &String {
    return &values[0];
}
```

compiler infers lifetime of return value is same as input

lifetime hard cases

```
// ERROR:
fn get_first_matching(prefix: &str, values: &Vec<String>)
                                              -> &String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}
```

this is a compile-error, because of the return value

compiler need to be told lifetime of return value

lifetime annotations

```
fn get_first_matching<'a, 'b>(prefix: &'a str, values: &'b Vec<String>)
                                    -> &'b String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}
```

prefix has lifetime a

values and returned string have lifetime b

lifetime annotations

```
fn get_first_matching<'a, 'b>(prefix: &'a str, values: &'b Vec<String>)
                              -> &'b String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}

fn get_first(values: &Vec<String>) -> &String {
    let prefix: String = compute_prefix();
    return get_first_matching(&prefix, values)
    // prefix deallocated here
}
```

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently
- object is "moved"

owner can let someone borrow object **temporarily**
- must know when object is given back

only **modify** object when exactly one user
- owner or exclusive borrower

restricting modification

```
fn modifyVector(vector: &mut Vec<u32>) { ... }
fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    for value in &vector {
        if value == 2 {
            modifyVector(&mut vector) // ERROR
        }
    }
}
```

trying to give away mutable reference

…while the for loop has a reference

what about dynamic allocation?

saw Rust's Vec class — equivalent to C++ vector/Java ArrayList

idea: Vec wraps a heap allocation of an array

owner of Vec "owns" heap allocation

- delete when no owner

also Box class — wraps heap allocation of a single value

- basically same as Vec except one element

escape hatch

Rust lets you avoid compiler's mechanisms

implement your own

unsafe keyword

how Vec is implemented

deep inside Vec

```
pub struct Vec<T> {
    buf: RawVec<T>, // interface to malloc
    len: usize,
}

impl<T> Vec<T> {
    ...
    pub fn truncate(&mut self, len: usize) {
        unsafe {
            // drop any extra elements
            while len < self.len {
                // decrement len before the drop_in_place(), so a panic on Drop
                // doesn't re-drop the just-failed value.
                self.len -= 1;
                let len = self.len;
                ptr::drop_in_place(self.get_unchecked_mut(len));
            }
        }
    }
    ...
}
```

Rust escape hatch support

escape hatch: make new reference-like types

callbacks on ownership ending (normally deallocation)

choice of what happens on move/copy

alternative rule: reference counting

keep track of number of references

delete when count goes to zero

- Rust automatically calls destructor — no programmer effort

Rust implement with Rc type (“counted reference”)

Ref Counting Example

```
struct Grade {
    score: i32, studentName: String, assignmentName: String,
}
struct Student {
    name: String,
    grades: Vec<Rc<Grade>>,
}
struct Assignment {
    name: String
    grades: Vec<Rc<Grade>>
}

fn add_grade(student: &mut Student, assignment: &mut Assignment, score: i32) {
    let grade = Rc::new(Grade {
        score: i32,
        studentName: student.name,
        assignmentName: assignment.name,
    })
    student.grades.push(grade.clone())
    assignment.grades.push(grade.clone())
}
```

Rust escape hatch support

escape hatch: make new reference-like types

Rc: `Rc<T>` acts like &T

callbacks on ownership ending (normally deallocation)

Rc: deallocating `Rc<T>` decrements shared count

choice of what happens on move/copy

Rc: transferring Rc makes new copy, increments shared count

Rc implementationed (annotated) (1)

```
impl<T: ?Sized> Clone for Rc<T> {
    ...
    fn clone(&self) -> Rc<T> {
        self.inc_strong(); // <-- incremenet reference count
        Rc { ptr: self.ptr }
    }
}
```

Rc implementation (annotated) (2)

```
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    ...
    fn drop(&mut self) { // <-- compilers calls on deallocation
        unsafe {
            let ptr = *self.ptr;

            self.dec_strong(); // <-- decrement reference cont
            if self.strong() == 0 { // if ref count is 0
                // destroy the contained object
                ptr::drop_in_place(&mut (*ptr).value);
                ...
             }
        }
    }
    ...
}
```

data races

Rusts rules around modification built assuming concurrency

OSes and other “systems programming” applications use multiple cores/threads

particular problem: value being used from multiple threads at same time

data races from use-after-free

given x: Rc<Foo> variable calling x.clone() on two cores

- some variable shared between two cores
- reference counting will prevent use-after-free, right?

```
x.clone on core A            x.clone on core B
--------------------------------------------

x.inc_strong():
  temp <- self.count
                             x.inc_strong():
                               temp <- self.count
                               self.count <- temp + 1
  self.count <- temp + 1
```

problem: reference count one too low!

Rust solution?

one option: require Rc implementation to handle mutiple cores

- problem: not zero overhead

Rust solution: different types for multithreaded/multicore code

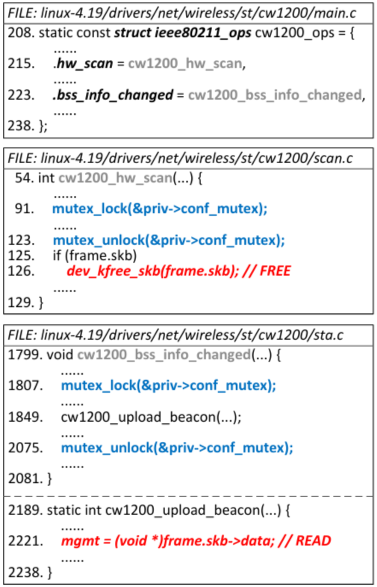
two “traits” to mark custom types:

- Sync: can be used from multiple cores/threads at once
- Send: can be moves from one thread to another

two implementations of referenc counting

- Rc: not suitable for multicore, not marked Sync/Send
- Arc: is suitable for multicore, slower than Rc probably

example: concurreny UAF bug



```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/main.c
208. static const struct ieee80211_ops cw1200_ops = {
        ......
215.    .hw_scan = cw1200_hw_scan,
        ......
223.    .bss_info_changed = cw1200_bss_info_changed,
        ......
238. };
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/scan.c
 54. int cw1200_hw_scan(...) {
        ......
 91.    mutex_lock(&priv->conf_mutex);
        ......
123.    mutex_unlock(&priv->conf_mutex);
125.    if (frame.skb)
126.       dev_kfree_skb(frame.skb); // FREE
        ......
129. }
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/sta.c
1799. void cw1200_bss_info_changed(...) {
         ......
1807.    mutex_lock(&priv->conf_mutex);
         ......
1849.    cw1200_upload_beacon(...);
         ......
2075.    mutex_unlock(&priv->conf_mutex);
         ......
2081. }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
2189. static int cw1200_upload_beacon(...) {
         ......
2221.    mgmt = (void *)frame.skb->data; // READ
         ......
2238. }
```

Figure 2: A reported bug in the *cw1200* driver in Linux 4.19

Figure from Bai, Lawall, Chen and Mu (Usenix ATC'19)

"Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux drivers"

bug in a wireless networking driver

other policies Rust supports

RefCell — borrowing, but check at runtime, not compile-time

- detect at runtime if used while already used
- internally: destructor call when returned object goes out of scope

Weak — reference-counting, but don't contribute to count

- detect at runtime if used with count $= 0$

Mutex — with multicore, enforce one user at a time by waiting

…

other policies Rust supports

RefCell — borrowing, but check at runtime, not compile-time

- detect at runtime if used while already used
- internally: destructor call when returned object goes out of scope

Weak — reference-counting, but don't contribute to count

- detect at runtime if used with count $= 0$

Mutex — with multicore, enforce one user at a time by waiting

…

zero-overhead

normal case — lifetimes — have no overhead

compiler proves safety, generates code with no bookkeeping

other policies (e.g. reference counting) do

...but can implement new ones if not good enough

other things languages can enforce?

saw: enforcing no use-after-free

lots of coding conventions we might try to enforce:

code's runtime does not depend on secret data

- secret data has different type
- variable time operations prohibited with secret data

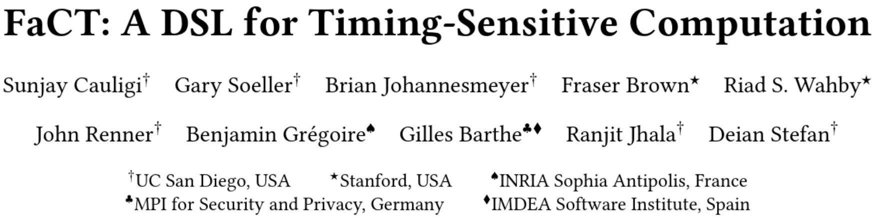
sensitive data not passed to wrong place

- sensitive data has different type
- assignment to wrong places is a type error

code has bounded runtime

- langauge prohibits not unbounded loops, recursion, etc.

some constant time ideas

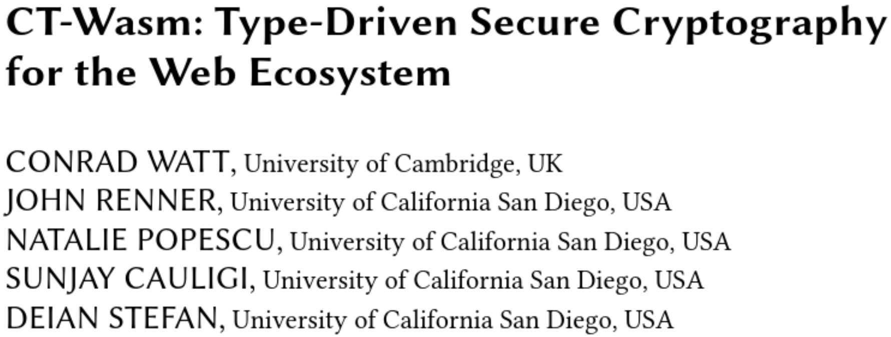


FaCT: A DSL for Timing-Sensitive Computation

Sunjay Cauligi[†] Gary Soeller[†] Brian Johannesmeyer[†] Fraser Brown[⋆] Riad S. Wahby[⋆]

John Renner[†] Benjamin Grégoire[♠] Gilles Barthe[♣♦] Ranjit Jhala[†] Deian Stefan[†]

[†]UC San Diego, USA [⋆]Stanford, USA [♠]INRIA Sophia Antipolis, France
[♣]MPI for Security and Privacy, Germany [♦]IMDEA Software Institute, Spain



CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem

CONRAD WATT, University of Cambridge, UK
JOHN RENNER, University of California San Diego, USA
NATALIE POPESCU, University of California San Diego, USA
SUNJAY CAULIGI, University of California San Diego, USA
DEIAN STEFAN, University of California San Diego, USA

FaCT, PLDI 2019; CT-Wasm: POPL 2019

constant-time programming languages

active research area, no consensus on what works best

common approach: separate type for **secret** data

compiler or language virtual machine disallows variable-time operations using secret data

no secret-based array lookup (cache timing varies)

e.g. array[secret_value] $\rightarrow$ compile error (type mismatch)

no secret-based integer division (usually variable speed instruction)

…

explicit operations for any secret-to-non-secret conversions

least privilege

a typical program I run on my desktop is allowed to...

make network connections to anywhere

upload all my files to the Internet

delete all my files

record all my keystrokes

...

but it probably doesn't need to...

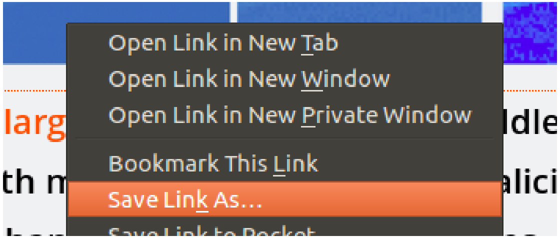
ideally: if typical program was compromised/malicious,
it still wouldn't be able to do most of these things

things applications need?

what things should browser be able to do?

what things should word processor be able to do?

things broswers need



save files

have your webmail password

...

multi-user OSs

```
cr4bd@labunix01:~$ cp myprogram.exe /bin/ls
cp: cannot create regular file ‘/bin/ls’: Permission denied
```

programs have limited privileges

permission enforcement

```
struct Process {
    int user_id;
    ...
};
int handle_open_system_call(char *filename, ...) {
    Process* currentProcess = GetCurrentProcess();
    File* file = GetFileByFilename(filename);
    if (!file->UserCanAccess(currentProcess->user_id)) {
        return ERROR_PERMISSION_DENIED;
    }
    ...
}
```

multi-user OSs

```
cr4bd@labunix01:~$ cp myprogram.exe /bin/ls
cp: cannot create regular file '/bin/ls': Permission denied
```

programs have limited privileges

OS tracks “user” of running every program

result: malware I installed shouldn’t be able to effect other users

idea 1: reuse this support for web browsers

- webpage should run as “different user”
- malware should only affect web browser?

the privilege separation idea

can't make whole browser run as "different user"

- still need to save files, read password, etc.

how about just the parts that are "dangerous"?

- part that runs scripts, parses HTML

simple privilege separation

simple example: want to show videos

video decoding library is tens of thousands of lines of code
- often buggy, includes hard-to-check hand-written assembly

what does video decoding library do?
- read video file as input
- output images as output

simple privilege seperation

setup: create new user

start video decoder as new user

communicate via “pipes”

- like terminal to be used by program

simple privilege seperation

```
/* dangerous video decoder to isolate */
int main() {
    /* switch to right user */
    SetUserTo("user-without-privileges"));
    while (fread(videoData, sizeof(videoData), 1, stdin) > 0) {
        doDangerousVideoDecoding(videoData, imageData);
        fwrite(imageData, sizeof(imageData), 1, stdout);
    }
}

/* code that uses it */
    FILE *fh = RunProgramAndGetFileHandle("./video-decoder");
    for (;;) {
        fwrite(getNextVideoData(), SIZE, 1, fh);
        fread(image, sizeof(image), 1, fh);
        displayImage(image);
    }
```

issues with privilege separation (1)

“other user” can still do too much

read unprotected files

  most of them?

write temporary files?

open network connections

use all your memory

…

issues with privilege separation (2)

awkward to do

switching users requires special permissions

seperate user for each video decoder, audio decoder, web page renderer?

    users can debug processes from same user

slowdown — extra copying

program to OS interface

primary way application talks to OS: system calls

function calls that request OS do something

typically: how program can interact with rest of system

- files
- other programs
- the network
- devices
- …

controlling program behavior: control what system calls

Linux system call filtering API

privilege seperation support: system call filtering

simple API: `seccomp(SECCOMP_SET_MODE_STRICT, 0, 0)`

"The only system calls the calling thread is permitted to make are `read`, `write`, `_exit`, and `sigreturn`. Other system calls [kill the program]."

read/write only work on already open files

later: what if we want to be finer-grained?

“sandboxing”

result of filtering operations called a “sandbox”

idea: attacker can play in sandbox as much as they want

can’t do anything “harmful”

other possible implementations:

- e.g. virtual machine

Chrome architecture

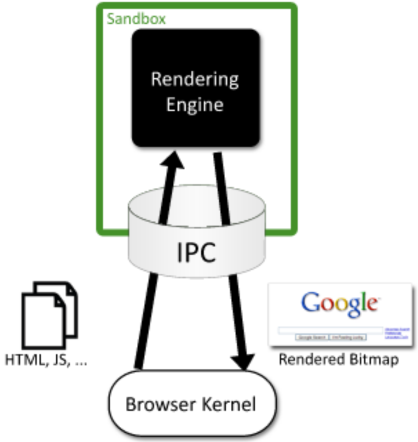


Figure 1: The browser kernel treats the rendering engine as a black box that parses web content and emits bitmaps of the rendered document.

talking to the sandbox

browser kernel sends commands to sandbox

sandbox sends commands to browser kernel

idea: commands only allow necessary things

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
    - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser “kernel” to sandbox

- send user input

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send

needs filtering — at least no `file:` (local file) URLs

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send user input

can still read any website!
still sends normal cookies!

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send user input

files go to download directory only
can't choose arbitrary filenames

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser “kernel” to sandbox

- send user input

browser kernel displays file choser
only permits files selected by user

Site Isolation

Chrome since version 67 (desktop)/77 (Mobile) has process per site

site ≈ registered domain name

complicated to implement:

- single web page can embed content from multiple sites
- web page can call serivces on other websites with “permission” of other website

OpenSSH privilege seperation

OpenSSH uses privilege seperation for its SSH server

what runs on the lab machines when you log into them

separate network processing code from authentication code

seperate process per connection — users don't share

OpenSSH privsep protocol

sandboxed process tells “monitor” to:

perform cryptographic operations

- long-term keys never in sandboxed process
- commands to ask for cryptographic messages they need

ask to switch to user — if given user password, etc.

- monitor process verifies login information

after authentication: new process running as logged-in user

- (normally) no issues with special privileges

privilege seperation overall

large application changes

- OpenSSH: 3k lines of code for communication/etc. added
- OpenSSH: 2% of existing code (950 of 44k lines) changed
- (but most changes simple)

lots of application knowledge

- what is a meaningful separation of 'privileged' and 'unprivileged'?

better application design anyways?

application confinement

confining whole browsers was hard

- we trust them to do a lot of things — e.g. write arbitrary files

but maybe we can do this for simpler applications?

idea 1: applications send system calls to OS

- limit syscalls like we limited browser kernel commands
- constructing command language “in reverse”

Linux system call filtering: detailed

Linux supports more fine-grained system call filtering

using BPF (Berkeley Packet Filter) programming language
- compiled in the kernel to assembly to check system calls

can check system call argument values, but...
- problems with pointer arguments
- too many system calls

Linux system call: open

```
open("foo.txt", O_RDONLY);
```

parameters:

- system call number: 2 (“open”)
- argument 1: 0x7fffe983 (address of string “foo.txt”)
- argument 2: 0 (value of “`O_RDONLY`”)

very problematic to filter using BPF interface

filtering system calls?

example: video player VLC playing a local file on my laptop

uses 73 unique kinds of system calls

opens many files that are not the video file

- libraries
- fonts
- configuration files
- translations of messages

can I limit the files my video player can read?

how do I come up with a useful filter?

Linux namespaces (1)

Linux: alternate sandboxing features

"namespaces" for ...

like partial virtual machine / "container"

example: run process in new mount (filesystem) namespace

- process has its own, limited view of available files

change to OS:

- each process has pointer to its view of available files
- some views have less stuff

Linux namespaces (2)

user namespace:

can run programs with new view of users:

inside namespace: running as root

outside namespace: root translated to innocent user ID

Linux namespaces (3)

user namespace and mount namespace together:

run program in new user + mount (filesystem) namespace

thinks it's running with full permissions

actually only able to access subset of files

...and only with permissions of limited user

Linux namespaces (4)

still need to figure out what files/etc. program needs

- “your entire documents directory” probably unsatisfying

OS code that enforces isolation is complicated

- much more likely to have bugs/omissions than system call filters
- a lot of work to maintain — is it worth it?

OS X sandboxing

OS X (tries to) implement system call filtering

main challenge: what about files?

- user can open a file anywhere — we expect that to work

OS X sandboxing

OS X (tries to) implement system call filtering

main challenge: what about files?

- user can open a file anywhere — we expect that to work

OS X solution: OS service displays file-open dialog

- OS knows user really choose a file

application can ask to remember file was chosen previously

not chosen/remembered — can't access

- requires changes to how applications open files

another sandboxing OS: Qubes

Qubes: heavily sandboxed OS

runs seperate VMs instead of filtering syscalls

UI that clearly shows what VM each window is from

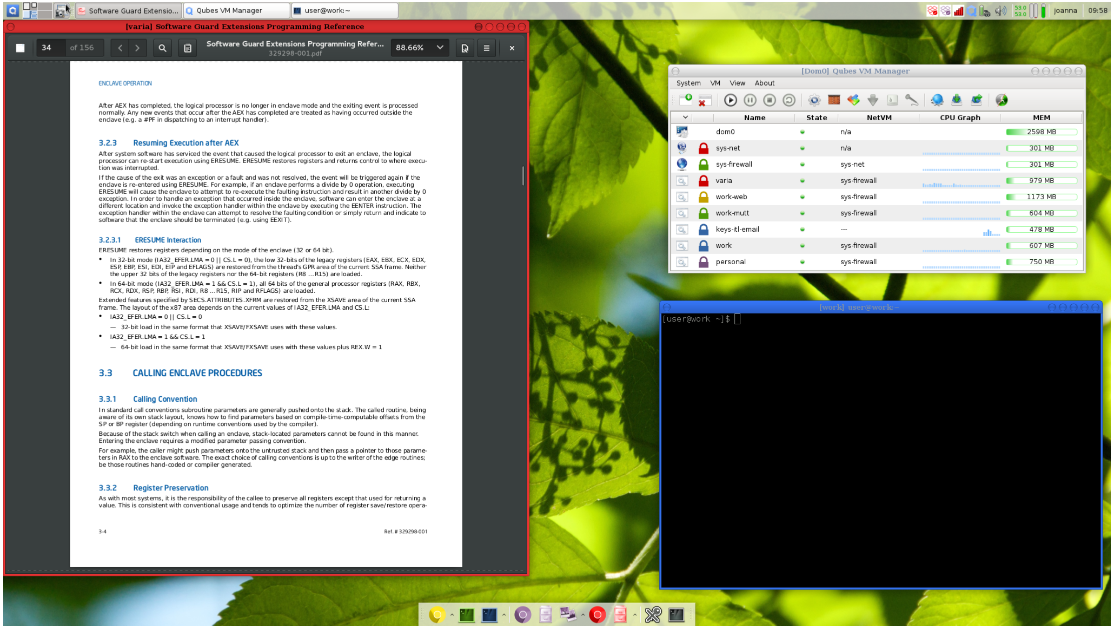
advantage: easier to gaurentee isolation

- many, many more bugs in system call filtering than VMs

disadvantage: harder to share between VMs

disadvantage: much more runtime overhead

Qubes screenshot


Software Guard Extensio...
Qubes VM Manager
user@work:~
joanna
09:58
[varia] Software Guard Extensions Programming Reference
34
of 156
Software Guard Extensions Programming Refer...
329298-001.pdf
88.66%
[Dom0] Qubes VM Manager
System
VM
View
About
[work] user@work:~

ENCLAVE OPERATION

After AEX has completed, the logical processor is no longer in enclave mode and the exiting event is processed normally. Any new events that occur after the AEX has completed are treated as having occurred outside the enclave (e.g. a #PF in dispatching to an interrupt handler).

3.2.3 Resuming Execution after AEX

After system software has serviced the event that caused the logical processor to exit an enclave, the logical processor can re-start execution using ERESUME. ERESUME restores registers and returns control to where execution was interrupted.

If the cause of the exit was an exception or a fault and was not resolved, the event will be triggered again if the enclave is re-entered using ERESUME. For example, if an enclave performs a divide by 0 operation, executing ERESUME will cause the enclave to attempt to re-execute the faulting instruction and result in another divide by 0 exception. In order to handle an exception that occurred inside the enclave, software can enter the enclave at a different location and invoke the exception handler within the enclave by executing the EENTER instruction. The exception handler within the enclave can attempt to resolve the faulting condition or simply return and indicate to software that the enclave should be terminated (e.g. using EEXIT).

3.2.3.1 ERESUME Interaction

ERESUME restores registers depending on the mode of the enclave (32 or 64 bit).

- In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32-bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP and EFLAGS) are restored from the thread's GPR area of the current SSA frame. Neither the upper 32 bits of the legacy registers nor the 64-bit registers (R8 ...R15) are loaded.
- In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 ...R15, RIP and RFLAGS) are loaded.

Extended features specified by SECS.ATTRIBUTES.XFRM are restored from the XSAVE area of the current SSA frame. The layout of the x87 area depends on the current values of IA32_EFER.LMA and CS.L:

- IA32_EFER.LMA = 0 || CS.L = 0
  — 32-bit load in the same format that XSAVE/FXSAVE uses with these values.
- IA32_EFER.LMA = 1 && CS.L = 1
  — 64-bit load in the same format that XSAVE/FXSAVE uses with these values plus REX.W = 1

3.3 CALLING ENCLAVE PROCEDURES

3.3.1 Calling Convention

In standard call conventions subroutine parameters are generally pushed onto the stack. The called routine, being aware of its own stack layout, knows how to find parameters based on compile-time-computable offsets from the SP or BP register (depending on runtime conventions used by the compiler).

Because of the stack switch when calling an enclave, stack-located parameters cannot be found in this manner. Entering the enclave requires a modified parameter passing convention.

For example, the caller might push parameters onto the untrusted stack and then pass a pointer to those parameters in RAX to the enclave software. The exact choice of calling conventions is up to the writer of the edge routines; be those routines hand-coded or compiler generated.

3.3.2 Register Preservation

As with most systems, it is the responsibility of the callee to preserve all registers except that used for returning a value. This is consistent with conventional usage and tends to optimize the number of register save/restore opera-

3-4 Ref. # 329298-001

Name	State	NetVM	MEM
dom0	●	n/a	2598 MB
sys-net	●	n/a	301 MB
sys-firewall	●	sys-net	301 MB
varia	●	sys-firewall	979 MB
work-web	●	sys-firewall	1173 MB
work-mutt	●	sys-firewall	604 MB
keys-itl-email	●	---	478 MB
work	●	sys-firewall	607 MB
personal	●	sys-firewall	750 MB

```
[user@work ~]$ 
```