The Case for Building a Kernel in Rust

Amit Levy† Brad Campbell‡ Branden Ghena‡
Pat Pannuto‡ Philip Levis† Prabal Dutta‡

Stanford University† & University of Michigan‡
Memory and type safety bugs plague systems
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2017-9996</td>
<td>The <code>cdx1_decode_frame</code> function in <code>libavcodec/cdx1.c</code> in FFmpeg 2.8.x before 2.8.12, 3.0.x before 3.0.8, 3.1.x before 3.1.8, 3.2.x before 3.2.5, and 3.3.x before 3.3.1 does not exclude the CHUNKY format, which allows remote attackers to cause a denial of service (heap-based buffer overflow and application crash) or possibly have unspecified other impact via a crafted file.</td>
</tr>
<tr>
<td>CVE-2017-9995</td>
<td><code>libavcodec/scpr.c</code> in FFmpeg 3.3 before 3.3.1 does not properly validate height and width data, which allows remote attackers to cause a denial of service (heap-based buffer overflow and application crash) or possibly have unspecified other impact via a crafted file.</td>
</tr>
<tr>
<td>CVE-2017-9994</td>
<td><code>libavcodec/webp.c</code> in FFmpeg before 2.8.12, 3.0.x before 3.0.8, 3.1.x before 3.1.8, 3.2.x before 3.2.5, and 3.3.x before 3.3.1 does not ensure that pix_fmt is set, which allows remote attackers to cause a denial of service (heap-based buffer overflow and application crash) or possibly have unspecified other impact via a crafted file, related to the vp8_decode_mb_row_no_filter and pred8x8_128_dc_8_c functions.</td>
</tr>
<tr>
<td>CVE-2017-9992</td>
<td>Heap-based buffer overflow in the <code>decode_dds1</code> function in <code>libavcodec/dfa.c</code> in FFmpeg before 2.8.12, 3.0.x before 3.0.8, 3.1.x before 3.1.8, 3.2.x before 3.2.5, and 3.3.x before 3.3.1 allows remote attackers to cause a denial of service (application crash) or possibly have unspecified other impact via a crafted file.</td>
</tr>
<tr>
<td>CVE-2017-9991</td>
<td>Heap-based buffer overflow in the <code>xwd_decode_frame</code> function in <code>libavcodec/xwddec.c</code> in FFmpeg before 2.8.12, 3.0.x before 3.0.8, 3.1.x before 3.1.8, 3.2.x before 3.2.5, and 3.3.x before 3.3.1 allows remote attackers to cause a denial of service (application crash) or possibly have unspecified other impact via a crafted file.</td>
</tr>
<tr>
<td>CVE-2017-9990</td>
<td>Stack-based buffer overflow in the <code>color_string_to_rgba</code> function in <code>libavcodec/xpmdec.c</code> in FFmpeg 3.3 before 3.3.1 allows remote attackers to cause a denial of service (application crash) or possibly have unspecified other impact via a crafted file.</td>
</tr>
<tr>
<td>CVE-2017-9987</td>
<td>There is a heap-based buffer overflow in the <code>hpel_motion</code> function in <code>mpegvideo_motion.c</code> in <code>libavcodec/12.1.1</code>. A crafted input can lead to a remote denial of service attack.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CVE-2017-9762</td>
<td>The <strong>cmd_info</strong> function in <strong>libr/core/cmd_info.c</strong> in radare2 1.5.0 allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and application crash) via a crafted binary file.</td>
</tr>
<tr>
<td>CVE-2017-9612</td>
<td>The <strong>Ins_IP</strong> function in <strong>base/ttinterp.c</strong> in Artex Ghostscript GhostXPS 9.21 allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and application crash) or possibly have unspecified other impact via a crafted document.</td>
</tr>
<tr>
<td>CVE-2017-9527</td>
<td>The <strong>mark_context_stack</strong> function in <strong>gc.c</strong> in mruby through 1.2.0 allows attackers to cause a denial of service (<strong>heap-based use-after-free</strong> and application crash) or possibly have unspecified other impact via a crafted .rb file.</td>
</tr>
<tr>
<td>CVE-2017-9520</td>
<td>The <strong>r_config_set</strong> function in <strong>libr/config/config.c</strong> in radare2 1.5.0 allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and application crash) via a crafted DEX file.</td>
</tr>
<tr>
<td>CVE-2017-9182</td>
<td><strong>libautotrace.a</strong> in AutoTrace 0.31.1 allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and invalid heap read), related to the <strong>GET_COLOR</strong> function in <strong>color.c:16:11</strong>.</td>
</tr>
<tr>
<td>CVE-2017-8929</td>
<td>The <strong>sized_string_cmp</strong> function in <strong>libyara/sizedstr.c</strong> in YARA 3.5.0 allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and application crash) via a crafted rule.</td>
</tr>
<tr>
<td>CVE-2017-8895</td>
<td>In Veritas Backup Exec 2014 before build 14.1.1187.1126, 15 before build 14.2.1180.3160, and 16 before FP1, there is a <strong>use-after-free</strong> vulnerability in multiple agents that can lead to a denial of service or remote code execution. An authenticated attacker can use this vulnerability to crash the agent or potentially take control of the agent process and then the system it is running on.</td>
</tr>
<tr>
<td>CVE-2017-8846</td>
<td>The <strong>read_stream</strong> function in <strong>stream.c</strong> in <strong>libr/zip.so</strong> in <strong>lrzip 0.631</strong> allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and application crash) via a crafted archive.</td>
</tr>
<tr>
<td>CVE-2017-8359</td>
<td>Google gRPC before 2017-03-29 has an out-of-bounds write caused by a heap-based <strong>use-after-free</strong> related to the <strong>grpc_call_destroy</strong> function in <strong>core/lib/surface/call.c</strong>.</td>
</tr>
<tr>
<td>CVE-2017-8270</td>
<td>In all Qualcomm products with Android releases from CAF using the Linux kernel, a race condition exists in a driver potentially leading to a <strong>use-after-free</strong> condition.</td>
</tr>
<tr>
<td>CVE-2017-8266</td>
<td>In all Qualcomm products with Android releases from CAF using the Linux kernel, a race condition exists in a video driver potentially leading to a <strong>use-after-free</strong> condition.</td>
</tr>
<tr>
<td>CVE-2017-7946</td>
<td>The <strong>get_relocs_64</strong> function in <strong>libr/bin/format/mach0/mach0.c</strong> in <strong>radare2 1.3.0</strong> allows remote attackers to cause a denial of service (<strong>use-after-free</strong> and application crash) via a crafted MachO file.</td>
</tr>
</tbody>
</table>
VULNERABILITY DETAILS
the value passed to function TwoByteSeqStringSetChar maybe not a smi but a HeapObject, simply casting a point to HeapObject to a smi lead to information leak.

```c
void FullCodeGenerator::EmitTwoByteSeqStringSetChar(CallRuntime* expr) {
  ZoneList<Expression*>* args = expr->arguments();
  DCHECK_EQ(3, args->length());

  Register string = rax;
  Register index = rbx;
  Register value = rcx;

  VisitForStackValue(args->at(0)); // index
  VisitForStackValue(args->at(1)); // value------> maybe point of heap object, i guess
  VisitForAccumulatorValue(args->at(2)); // string
  PopOperand(value);
  PopOperand(index);
```
VULNERABILITY DETAILS

the value passed to function TwoByteSeqStringSetChar maybe not a smi but a HeapObject,
simply casting a point to HeapObject to a smi lead to information leak.

void FullCodeGenerator::EmitTwoByteSeqString:
  ZoneList<Expression>* args = expr->argument;
  DCHECK_EQ(3, args->length());

  Register string = rax;
  Register index = rbx;
  Register value = rcx;
  i 
  VisitForStackValue(args->at(0)); //
  VisitForStackValue(args->at(1)); //
  VisitForAccumulatorValue(args->at(2)); //
  PopOperand(value);
  PopOperand(index);

MsMpEng is the Malware Protection service that is enabled by default on Windows 8, 8.1, 10, Windows Server 2012, and so on. Additionally, Microsoft Security Essentials, Endpoint Protection and various other Microsoft security products share an engine. MsMpEng runs as NT AUTHORITY\SYSTEM without sandboxing, and without authentication via various Windows services, including Exchange.

On workstations, attackers can access Mpengine by sending emails to or opening attachments is not necessary), visiting links in a web browser and so on. This level of accessibility is possible because MsMpEng is a minifilter to intercept and inspect all system filesystem activity, contents to anywhere on disk (e.g. caches, temporary internet files, unconfirmed downloads), attachments, etc) is enough to access functions. MIME types and file extensions are not relevant to this vulnerability.

Vulnerabilities in MsMpEng are among the most severe possible in Windows privilege, accessibility, and ubiquity of the service.

The core component of MsMpEng responsible for scanning and analysis. Mpengine is a vast and complex attack surface, comprising of handled archive formats, executable packers and cryptors, full system emulation, ...
VULNERABILITY DETAILS
the value passed to function TwoByteSeqStringSetChar maybe not a smi but a HeapObject,
simply casting a point to HeapObject to a smi lead to information leak.

```c
void FullCodeGenerator::m_EmitTwoByteSeqString:
    ZoneList<Expression*>* args = expr->argument;
    DCHECK_EQ(3, args->length());
    Register string = rax;
    Register index = rbx;
    Register value = rcx;
    VisitForStackValue(args->at(0));  // i guess
    VisitForStackValue(args->at(1));  // i guess
    VisitForAccumulatorValue(args->at(2));  // i guess
    PopOperand(value);
    PopOperand(index);
```

**MsMpEng: Remotely Exploitable Type Confusion in Windows 10, Windows Server, SCEP, Microsoft Security Essentials, and Exchange**

MsMpEng is the Malware Protection service that is enabled by default on Windows 8, 8.1, 10, Windows Server 2012, and so on. Additionally, Microsoft Security Essentials, Endpoint Protection and various other Microsoft security products share the same engine. MsMpEng runs as NT AUTHORITY\SYSTEM without sandboxing, and without authentication via various Windows services, including Exchange.

On workstations, attackers can access mpengine by sending emails to or opening attachments is not necessary), visiting links in a web browser, and so on. This level of accessibility is possible because MsMpEng consumes system filesystem activity, processes, temporary internet files (cache and cookies, etc) is enough to access functions relevant to this vulnerability.

The most severe possible in Windows is the remote execution of the service.

```html
POC:

<!DOCTYPE html>
<html>
<body onload='setInterval(boom, 100)'>

<style>
  .class1 { float: left; column-count: 5; }
  .class2 { column-span: all; columns: 1px; }
  table { border-spacing: 0px; }
</style>

<table cellspacing="0">
  <tr class="class1">
    <th id="th1" colspan="5" width=100></th>
    <th colspan="5" width=50></th>
  </tr>
</table>

<script>
  function boom() {
    document.styleSheets[0].media.mediaText = "aaaaaaaaaaaaaaaaaaa";
    th1.align = "right";
  }
</script>

Note: The analysis below is based on an 64-bit IE (running in single process mode) running on Windows Server Symbol Server has been down for several days and that's the only configuration for which I had up-to-date Edge and 32-bit IE 11 should behave similarly.

The PoC crashes in
Long history of research

“Bug finding”
- Fuzz-ing (1990)
- DART (2005)
- KLEE (2008)
- KINT (2012)

Type-Safe Kernels:
- Cedar (1986)
- Spin (1995)
- Singularity (2007)
Why are we still building systems in C?
Type safety (typically) isn’t free

Type safety usually requires garbage collection.

- Give up control over memory layout and location
- Large trusted runtime
- Either a performance hit or large memory overhead
Type safety (typically) isn’t free

Type safety usually requires garbage collection.

- Give up control over memory layout and location
- Large trusted runtime
- Either a performance hit or large memory overhead

Can we use type safety without allowing it to dictate how we design systems?
“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.”

-https://www.rust-lang.org

1. A 5-minute Introduction to Rust
2. Limitations imposed by Rust
3. Addressing the limitations
4. Case-Study: Tock OS
5. Conclusion & future work
A Systems Builder’s Guide to Rust (abridged)
Rust Features

- Type and memory safe
- Statically enforced type system
- Compiles with the LLVM toolchain to machine code
- C calling convention
- Explicit memory location and layout
- No language runtime
Ownership

Key Property

When the owner goes out of scope, we can deallocate memory for the value.
Ownership

Key Property

When the owner goes out of scope, we can deallocate memory for the value.

Memory for the value Foo::new() is allocated and bound to the variable x.

```
{  
    let x = Foo::new()
}
```

When the scope exits, x is no longer valid and the memory is “freed”
This is an error:

```rust
{
    let x = Foo::new();
    let y = x;
    // x not valid here
}
```

because `Foo::new()` has been moved from `x` to `y`, so `x` is no longer valid.
fn bar(x: &mut Foo) {
    // the borrow is implicitly released.
}

let mut x = Foo::new();
bar(&mut x);
// x still valid here
fn bar(x: &mut Foo) {
    // the borrow is implicitly released.
}

let mut x = Foo::new();
bar(&mut x);
// x still valid here

Just a pointer at runtime
fn bar(x: &mut Foo) {
    // the borrow is implicitly released.
}

let mut x = Foo::new();
bar(&mut x);
// x still valid here

Just a pointer at runtime

- Mutable references (&mut) must be unique
- Shared references (&) cannot mutate the value
enum NumOrPointer {
    Num(u32),
    Pointer(&mut u32)
}
enum NumOrPointer {  
    Num(u32),  
    Pointer(&mut u32)  
}

// n.b. will not compile
let external : &mut NumOrPointer;
if let Pointer(internal) = external {
    *external = Num(0xdeadbeef);
    *internal = 12345;
    // Kaboom: we've just written '12345'
    // to the address '0xdeadbeef'
}
enum NumOrPointer {
    Num(u32),
    Pointer(&mut u32)
}

// n.b. will not compile
let external: &mut NumOrPointer;
if let Pointer(internal) = external {
    *external = Num(0xdeadbeef);
    *internal = 12345;
    // Kaboom: we've just written '12345'
    // to the address '0xdeadbeef'
}
enum NumOrPointer {
    Num(u32),
    Pointer(&mut u32)
}

// n.b. will not compile
let external : &mut NumOrPointer;
if let Pointer(internal) = external {
    *external = Num(0xdeadbeef);
    *internal = 12345;
    // Kaboom: we’ve just written ‘12345’
    // to the address ‘0xdeadbeef’
}
```rust
enum NumOrPointer {
    Num(u32),
    Pointer(&mut u32)
}

// n.b. will not compile
let external : &mut NumOrPointer;
if let Pointer(internal) = external {
    *external = Num(0xdeadbeef);
    *internal = 12345;
    // Kaboom: we’ve just written ‘12345‘
    // to the address ‘0xdeadbeef‘
}

$ rustc test.rs
error[E0506]: cannot assign to `external`
    because it is borrowed
```
Rust imposed limitations
pub struct SysCallDispatcher {
    processes: Vec<Process>,
    pool: &mut RandomPool,
    ...
}

pub struct RandomPool {
    busy: bool,
    pool: Queue<u32>,
    rng: &mut RNG,
    syscall: &mut SysCallDispatcher,
}

pub struct RNG {
    hw_registers: [usize; 16],
    client: &mut RandomPool,
}
let syscall: SysCallDispatcher;
let pool: RandomPool;
let rng: RNG;

syscalls.pool = &mut pool;
pool.syscall = &mut syscall;
pool.rng = &mut rng;
rng.client = &mut pool;
It’s actually safe to have mutable aliases in many cases.

The key is avoiding mutability and aliasing simultaneously.
It’s actually safe to have mutable aliases in *many cases*. The key is avoiding mutability and aliasing simultaneously. Rust has container types with “interior mutability”. Shared references to these types allow mutation, give certain restrictions:

- **Cell**: Only copy-in/out or replace, no references to internal value
- **Mutex**: Gives internal references through mutual-exclusion
- **TakeCell**: Only operates if not already being used
pub struct SysCallDispatcher {
    processes: TakeCell<Vec<Process>>,
    pool: &RandomPool,
    ...
}

pub struct RandomPool {
    busy: Cell<bool>,
    pool: TakeCell<Queue<u32>>,
    rng: &RNG,
    syscall: &SysCallDispatcher,
}

pub struct RNG {
    hw_registers: TakeCell<[usize; 16]>,
    client: &RandomPool,
}
let syscall: SysCallDispatcher;
let pool: RandomPool;
let rng: RNG;

syscalls.pool = &pool;
pool.syscall = &syscall;
pool.rng = &rng;
rng.client = &pool;
Case study: Tock OS
Security focused embedded operating system
Tock Overview

- Security focused embedded operating system
- Kernel components are mostly untrusted
Tock Overview

- Security focused embedded operating system
- Kernel components are mostly untrusted
- Targets microcontrollers with <64kB RAM
Kernel written in ~26695 lines of Rust
Kernel written in ~26695 lines of Rust
struct DMAChannel {
    ...
    enabled: Cell<\texttt{bool}>,
    buffer: TakeCell<&\texttt{\textquote{n}}\texttt{static mut [u8]}>,
}
```rust
enum EpCtl {
    ...,
    Enable = 1 << 31,
    ClearNak = 1 << 26,
    Stall = 1 << 21
}

struct InEndpoint {
    control: Cell<EpCtl>,
    dma_address: Cell<&'static DMADescriptor>,
    ...,
}

struct USBRegisters {
    ...,
    in_endpoints: Cell<&[InEndpoint; 16]>,
}
```
Minimal TCB

Trusted Kernel components (~3600 LoC)

- Board configuration: 806 LoC
- Process scheduler: 1784 LoC
- Hardware interface: ~1000 LoC

Rust core library

- Cell
- String, slice
- Floating point
- Compiler intrinsics (e.g. memcpy)
Conclusion
Type safety critical tool for building secure systems.
Type safety critical tool for building secure systems.

Key question: What is the lowest level of control needed?
Type Safety Belongs Everywhere

- Type safety critical tool for building secure systems.

- Key question: What is the lowest level of control needed?

- *Rust* is an existence proof we can use *today*.
Type Safety Belongs Everywhere

- Type safety critical tool for building secure systems.
- Key question: What is the lowest level of control needed?
- Rust is an existence proof we can use today
- We still need useful performance metrics

amit@amitlevy.com
https://tockos.org/
@talkingtock
Type Safety Belongs Everywhere

- Type safety critical tool for building secure systems.
- Key question: What is the lowest level of control needed?
- Rust is an existence proof we can use today
- We still need useful performance metrics
- Should experiment with other system types
Type safety critical tool for building secure systems.

Key question: What is the lowest level of control needed?

*Rust* is an existence proof we can use *today*

We still need useful performance metrics

Should experiment with other system types

Can we provide rich security primitives like DIFC?
Type Safety Belongs Everywhere

- Type safety critical tool for building secure systems.
- Key question: What is the lowest level of control needed?
- *Rust* is an existence proof we can use *today*
- We still need useful performance metrics
- Should experiment with other system types
- Can we provide rich security primitives like DIFC?

amit@amitlevy.com
https://tockos.org/
@talkingtock
Type Safety Belongs Everywhere

- Type safety critical tool for building secure systems.
- Key question: What is the lowest level of control needed?
- Rust is an existence proof we can use today
- We still need useful performance metrics
- Should experiment with other system types
- Can we provide rich security primitives like DIFC?

amit@amitlevy.com
https://tockos.org/
@talkingtock
struct App {
    count: u32,
    tx_callback: Callback,
    rx_callback: Callback,
    app_read: Option<AppSlice<Shared, u8>>,
    app_write: Option<AppSlice<Shared, u8>>,
}

pub struct Driver {
    app: TakeCell<App>,
}

driver.app.map(|app| {
    app.count = app.count + 1
});
/* Load App address into r1, replace with null */
ldr r1, [r0, 0]
movs r2, 0
str r2, [r0, 0]
/* If TakeCell is empty (null) return */
cmp r1, 0
it eq
bx lr
/* Non-null: increment count */
ldr r2, [r1, 0]
add r2, r2, 1
str r2, [r1, 0]
/* Store App back to TakeCell */
str r1, [r0, 0]
bx lr