presented by

Microsoft

UEFI 2025 Developers Conference & Plugfest

October 9, 2025

Michael Kubacki

Meet the Presenters

Michael Kubacki
Principal Software Engineer

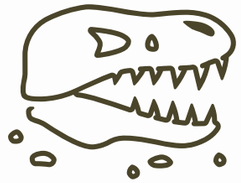
- Michael Kubacki is a firmware engineer in Microsoft's Core UEFI team. He serves as a steward in the Tianocore organization and Patina Project Chair in the Open Device Partnership organization.

- He is an advocate of open-source firmware, specializing in Intel platforms with a current focus on FW/OS interaction, Management Mode (MM) security, and Rust firmware.

Agenda

- Memory Safety
- Rust in Firmware
- The Patina Project
- Next Steps

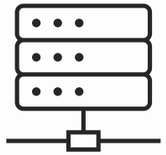
Firmware Language Evolution



Legacy BIOS

UEFI Standard

Converged Firmware Core



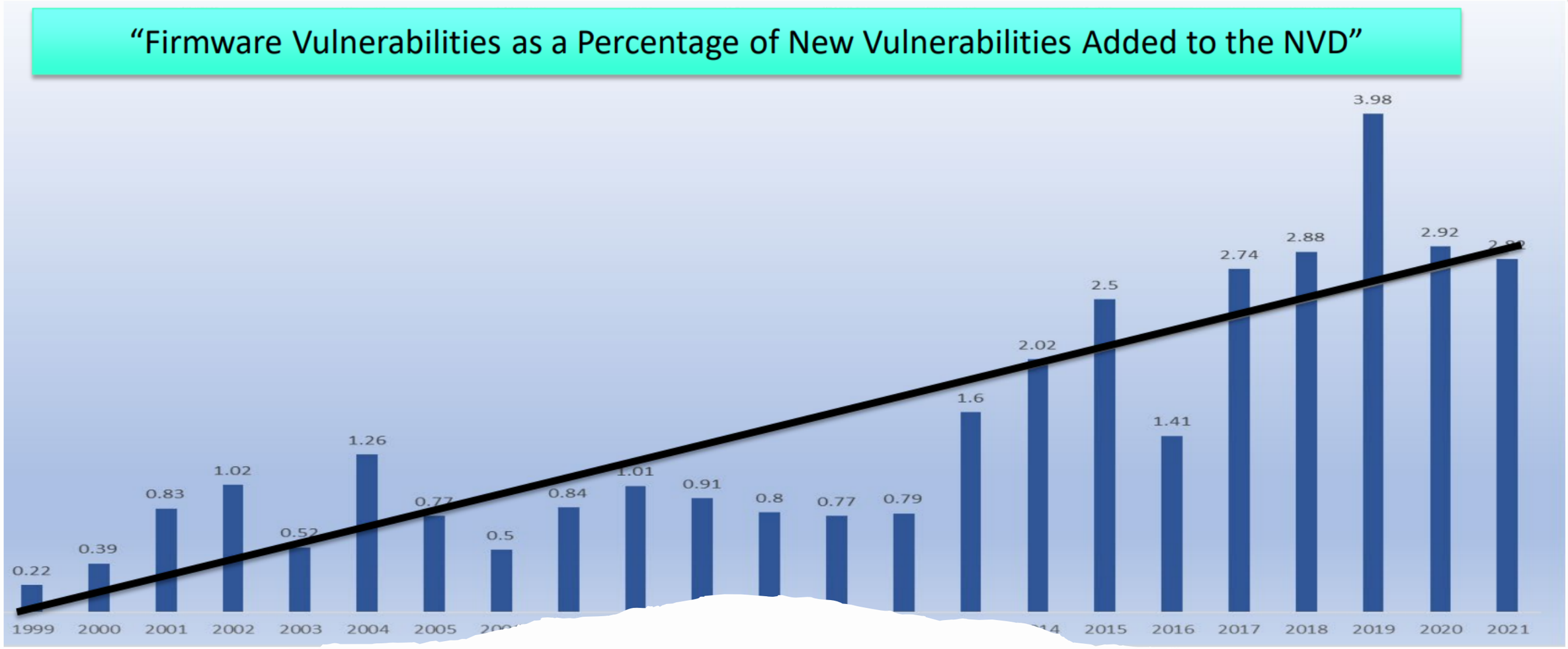
Segment Scaling

Memory Safe Firmware

1980 2004 2009 2015 Now

Assembly C Rust

Firmware is An Attractive Target



Source: [DHS CISA Strategy to Fix Vulnerabilities Below the OS Among Worst Offenders]

UEFI Plugfest 2020…

Top Issue	Open Source	Close Source
Buffer Overflow/ Integer Overflow	50%	38%
SMM	7%	18%
Variable	8%	5%
Register Lock	3%	10%

Summary & Call for Action

- 50% of EDKII security issues are memory issues.
- RUST can help to mitigate memory issues.
- Write critical firmware modules in RUST.

Rather than providing guidance and tools for addressing flaws,

we should strive to prevent the developer from introducing the flaws in the first place.

UEFI 2020 Virtual Plugfest - Enabling RUST for UEFI Firmware

Rust?

Build it in Rust

In 2018, the Rust community decided to improve the programming experience for a few distinct domains (see the 2018 roadmap). For these, you can find many high-quality crates and some awesome guides on how to get started.

Command Line

WebAssembly

Networking

Embedded

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity

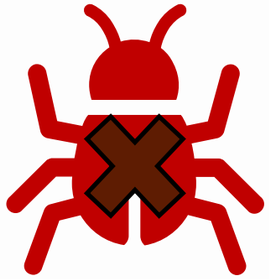
Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Rust Programming Language

Benefits of Rust in Firmware

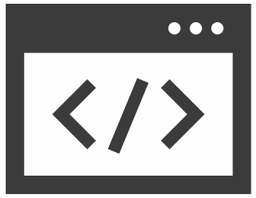
Performant & Reliable

Memory Safety Without Garbage Collection



Stringent Type Safety and Concurrency Rules Prevent Subtle Bugs

Exposes a Broader Community with a Built-In Package Manager



Developer Productivity Enhancements (e.g. Generic & Traits)

Better Tooling w/ Built-In Docs

Scores High in Developer Satisfaction

Is Memory Safety Really an Issue?

Yes.

CVE ID	CVSS Score	Vulnerability Type	Potential Rust Prevention Mechanism
CVE-2023-45230	8.3 (HIGH)	Buffer Overflow in DHCPv6	Automatic slice bounds checking
CVE-2022-36765	7.0 (HIGH)	Integer Overflow in CreateHob()	Checked arithmetic operations
CVE-2023-45229	6.5 (MEDIUM)	Out-of-Bounds Read in DHCPv6	Slice bounds verification
CVE-2014-8271	6.8 (MEDIUM)	Buffer Overflow in Variable Processing	Dynamic Vec sizing eliminates fixed buffers
CVE-2023-45233	7.5 (HIGH)	Infinite Loop in IPv6 Parsing	Iterator patterns with explicit termination
CVE-2021-38575	8.1 (HIGH)	Remote Buffer Overflow in iSCSI	Slice-based network parsing with bounds checking
CVE-2019-14563	7.8 (HIGH)	Integer Truncation	Explicit type conversions with error handling
CVE-2024-1298	6.0 (MEDIUM)	Division by Zero from Integer Overflow	Checked arithmetic prevents overflow-induced division by zero
CVE-2014-4859	6.8 (MEDIUM)	Integer Overflow in Capsule Update	Safe arithmetic with explicit overflow checking

Memory Safety in Rust

Rust keeps a focus on safety and speed with “zero-cost abstractions.” Moves safety checks to compile-time. Rust programmers think about memory “ownership.”

Key feature: Ownership

- Each value in Rust has an owner
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

```
let v = vec![1, 2, 3];
let v2 = v;

println!("v[0] is: {}", v[0]);
```

Rust Playground

```
warning: unused variable: `v2`
 --> src/main.rs:3:7
  |
3 |     let v2 = v;
  |         ^^ help: if this is intentional, prefix it with an underscore: `_v2`
  |
  = note: `#[warn(unused_variables)]` on by default

error[E0382]: borrow of moved value: `v`
 --> src/main.rs:5:27
  |
2 |     let v = vec![1, 2, 3];
  |         - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
3 |     let v2 = v;
  |              - value moved here
4 |
5 |     println!("v[0] is: {}", v[0]);
  |                             ^ value borrowed here after move
  |
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let v2 = v.clone();
  |               ++++++++

For more information about this error, try `rustc --explain E0382`.
```

Memory Safety in Rust

Rust keeps a focus on safety and speed with “zero-cost abstractions”. Moves safety checks to compile-time. Rust programmers work with the “borrow checker”.

Key feature: Borrowing (References)

- Any borrow must last for a scope no greater than that of the owner
- You may have one or the other of these two kinds of borrow, but not both at the same time:
  - One or more immutable references (&T) to a resource
  - Exactly on mutable reference (&mut T) to a resource

```
let mut x = 1;
let y = &mut x;
let z = &x;


*y += 1;


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

```
error[E0502]: cannot borrow `x` as immutable because it is also borrowed as mutable
 --> src/main.rs:4:13
  |
3 |     let y = &mut x;
  |             ------ mutable borrow occurs here
4 |     let z = &x;
  |             ^^ immutable borrow occurs here
5 |
6 |     *y += 1;
  |     ------- mutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
```

Rust Playground

Memory Safety in Rust

Rust keeps a focus on safety and speed with “zero-cost abstractions.” Moves safety checks to compile-time. Rust programmers think about “lifetimes.”

Key feature: Lifetimes

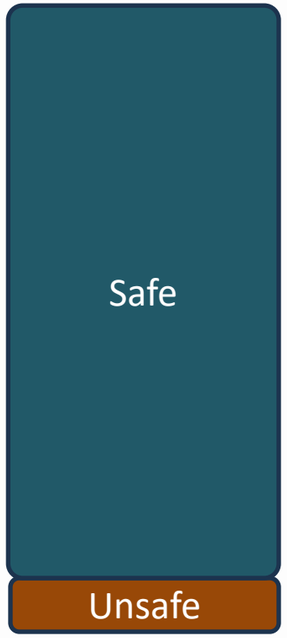
- Named regions of code that a reference must be valid for
- A lifetime often coincides with scope
- When the owner goes out of scope, the value will be dropped

```
fn main() {
    // Declare r without initializing
    let r;                     {
        // x lives only in this scope
        let x = 5;
        // Try to borrow x
        r = &x;
    }   // x goes out of scope here!
    // r cannot be used
    println!("{}", r);
}
```

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:13
  |
4 |         let x = 5;          // x lives only in this scope
  |             - binding `x` declared here
5 |         r = &x;             // Try to borrow x
  |             ^^ borrowed value does not live long enough
6 |     }                       // x goes out of scope here!
  |     - `x` dropped here while still borrowed
7 |     println!("{}", r);      // r cannot be used
  |                    - borrow later used here

For more information about this error, try `rustc --explain E0597`.
```

Rust Playground

How is This Different Than C?



In Rust safety is enforced by the compiler at the language level

- Memory safety violations are **compile-time errors**, not runtime bugs
- There is clear (and enforced) separate between *safe* and *unsafe* code using the `unsafe` keyword
- Safe abstractions are guaranteed safe by the compiler, not developer promises
- Unsafe code has strict requirements, clear documentation, and caller obligations that are compiler and lint enforced

The compiler **guarantees** that safe code cannot:

- Access memory out of bounds
- Use memory after it's freed
- Have data races in multi-threaded code
- Dereference null or dangling pointers

```
// Safe code - the compiler guarantees memory safety
let mut buffer = Vec::new();            // Dynamic allocation
buffer.extend_from_slice(user_input);  // Automatic bounds checking
let value = buffer[0];                  // Automatic bounds checking
let safe_value = buffer.get(0);         // Returns Option<T> - no panic
```

```
// Ownership prevents use-after-free at compile time
let data = vec![1, 2, 3];
let reference = &data[0];
drop(data);                    // COMPILE ERROR: cannot drop while borrowed
println!("{}", reference);  // This line would never be reached
```

Unsafe?

The goal is to write safe abstractions for operations that must be unsafe. What must be unsafe?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of a union

Note: Unsafe does not mean the code is bad. It gives access to the five features above. It does not disable the borrow checker or any of Rust's other safety checks. For example, a reference in unsafe code is still checked.

Unsafe puts certain constraints in place. For example, a function marked unsafe must be called from an unsafe block. This forces the caller to acknowledge the safety requirements and identifies area of unsafe usage in the codebase.

In Rust, all code is safe unless marked as unsafe.

```
// Unsafe functions must declare their safety requirements
/// # Safety
///
/// The caller must ensure:
/// - `ptr` is valid for reads of `size` bytes
/// - `ptr` is properly aligned for type T
/// - The memory referenced by `ptr` is not mutated during this call
/// - The memory referenced by `ptr` contains a valid value of type T
unsafe fn read_unaligned<T>(ptr: *const u8, size: usize) -> T {
    // Implementation that bypasses compiler safety checks
    std::ptr::read_unaligned(ptr as *const T)
}
```

Initial Challenges

Choosing Investments

- Considering ongoing deliverables and long-term Return on Investment (ROI)

Defining Architecture and Code Organization

- How to most effectively leverage Rust in firmware?
- Rewrite major core components in Rust?
  - Improve the underlying (language agnostic) architecture as well?
- Where do we work with others to design reusable code? What do we share?
- When can we ship meaningful Rust code?

Bootstrapping Rust in UEFI

- Adding Rust support in UEFI firmware tools & processes
  - Build, debug, …
- Training developers, integrating into platform firmware
- Working with other projects and adding foundational support for UEFI Rust
- Getting performance and size to be acceptable for production platforms

Manage the whole project in open-source.

Approaches

1. Build EFI Applications against the Rust target

2. Integrate Rust build support into the EDK II build system referencing tianocore/edk2-staging at edkii-rust.
   - Writing UEFI modules in a mix of C and Rust.

     See: UEFI 2020 Virtual Plugfest - Enabling RUST for UEFI Firmware

3. Build Pure Rust code into a binary that is integrated into firmware.

Patina

Open Device Partnership (ODP)

ODP

(GitHub org)

A new GitHub organization called

Open Device Partnership

Enhanced Security

Security threats continue to evolve. ODP takes a proactive approach: reducing attack surfaces, using secure hardware features, leveraging the memory-safe Rust language, and designing every component with security-first principles.

Standardization

Many device firmware components are "invisible plumbing" — necessary but costly to build and maintain. ODP's standards-based approach simplifies this infrastructure, maximizing reuse across devices, architectures (x86 and ARM), and generations.

Accelerated Development

Open collaboration means sharing solutions, reducing duplicated work, and speeding up the development of high-quality products.

The Patina Project

Patina is a project in Open Device Partnership (ODP) to modernize UEFI firmware development.

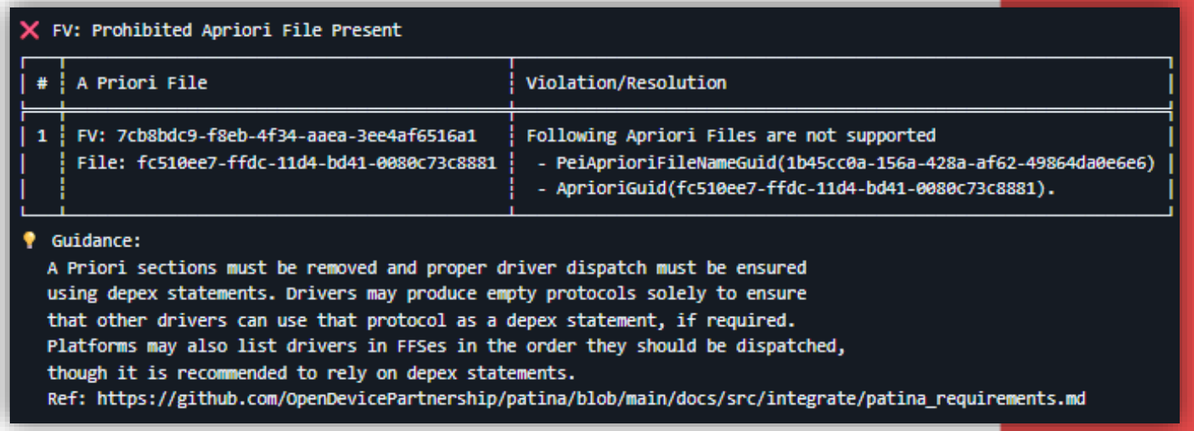
- Eliminate problematic features and patterns
- Support Enhanced Memory Protections for UEFI Firmware from the start
- Reuse more code from the open-source community and other projects
- Require > 80%-unit test coverage in all modules
- Reduce/remove firmware-specific build infrastructure
- Develop a platform integration model that can support a roadmap to replace Platform Initialization (PI)/Firmware File System (FFS)-based dispatch with pure Rust code

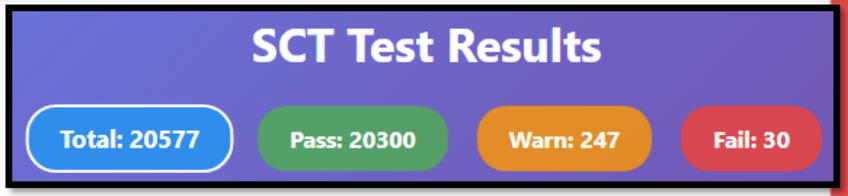
The main feature right now is the
“Patina Driver Execution Environment (DXE) Core.”

Notable Patina DXE Core Features

- AARCH64 and x86/64 support
  - Support for Quick Emulator (QEMU) (Q35 & SBSA)
  - Tested and developed on physical Intel and Arm hardware
  - Boots to Linux and Windows on these platforms
- Built-in decompression
- Elimination of problematic UEFI firmware features
- Infrastructure for on-platform testing
- Page table management
  - Complies with Microsoft Enhanced Firmware Memory Protections
- Pure Rust dispatch system (“components”) in addition to support for PI compatible FV/FFS dispatch (ability to dispatch today’s C drivers)
- UEFI performance tracing support
- UEFI source level debugging support (WinDbg)
- ~80% unit test coverage

A “Patina Readiness Tool” (written in Rust) identifies changes vendors need to make to use the Patina DXE Core.



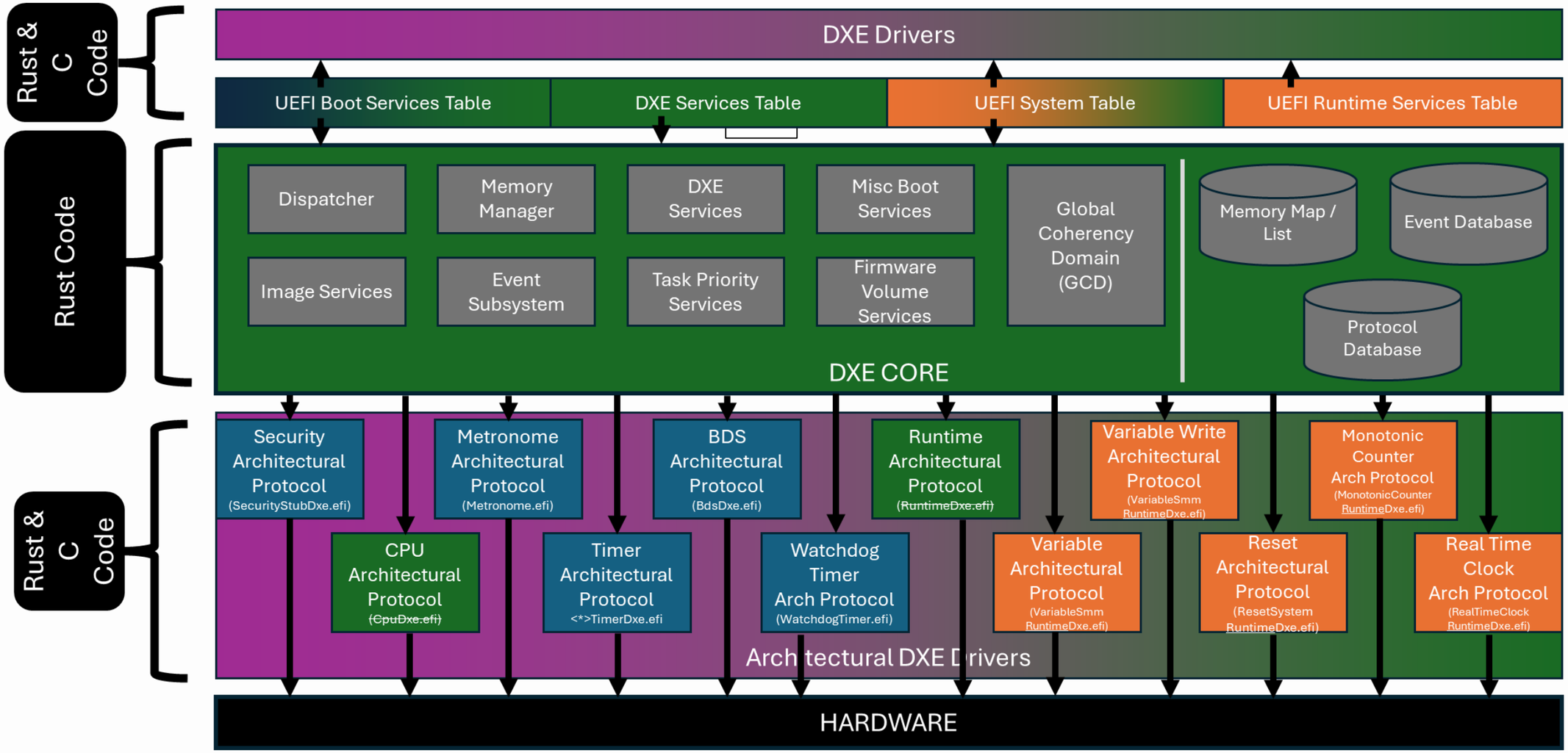


~10 failures are driver logic issues unrelated to Patina

Notable Patina DXE Core Requirements

1. No Traditional System Management Mode (SMM) Support
   - The following Firmware File System (FFS) file types are not dispatched:
     - EFI_FV_FILETYPE_SMM (0xA)
     - EFI_FV_FILETYPE_SMM_CORE (0xD)
     - EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER (0x8)
     - EFI_FV_FILETYPE_COMBINED_SMM_DXE (0xC)
2. A Priori Driver Dispatch is Not Supported
3. DXE Driver Sections Must Be Aligned to 4k
4. Platform Reported Resources Cannot Overlap
5. Page 0 Cannot Be Allocated (used for Null Pointer Detection)
6. MMIO and Reserved Resources Must Have Caching Attributes Specified (using a Resource Descriptor V2 Hand-Off Block)

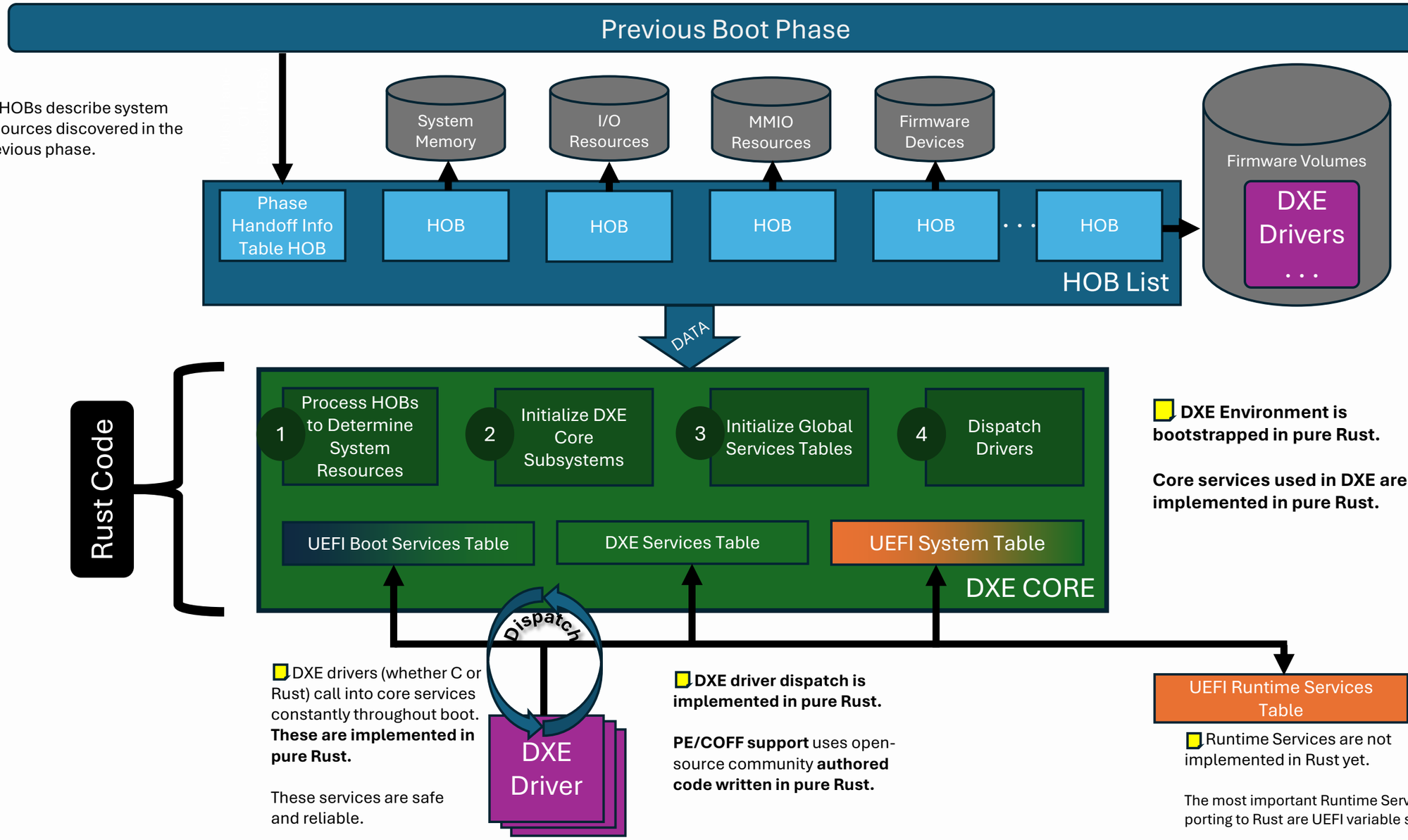
Transitioning the DXE Core to Rust



Legend

- C Driver (Boot Service or Runtime)
- C Boot Service Entity
- Rust Driver/Entity (Boot Service or Runtime)
- C Runtime Service Entity
- Rust DXE Core Subsystem

Patina DXE Core Initialization



Current Rust Boot Footprint

Essential services invoked most frequently throughout DXE boot phase are implemented in Rust.

Type	Service	Implemented in Pure Rust	Call Count (QEMU X64)
Driver Support	ConnectController()	Yes	517
	DisconnectController()	Yes	0
Event	CheckEvent()	Yes	27,347
	CloseEvent()	Yes	2,082
	CreateEvent()	Yes	2,153 (combined with ex())
	CreateEventEx()	Yes	2,153
	SetTimer()	No (Depends on Timer Arch Protocol)	4,063
	SignalEvent()	Yes	230,045
	WaitForEvent()	Yes	0
Image	Exit()	Yes	133
	LoadImage()	Yes	132
	StartImage()	Yes	133
	UnloadImage()	Yes	0
Memory	AllocatePages()	Yes	1,127
	AllocatePool()	Yes	19,696
	CopyMem()	Yes	Not Measured
	FreePages()	Yes	801
	FreePool()	Yes	14,763
	GetMemoryMap()	Yes	46
	SetMem()	Yes	Not Measured
Miscellaneous	CalculateCrc32()	Yes	440
	ExitBootServices()	Yes	2
	InstallConfigurationTable()	Yes	44

Type	Service	Implemented in Pure Rust	Call Count (QEMU X64)
Protocol	CloseProtocol()	Yes	544
	HandleProtocol()	Yes	25,915
	InstallMultipleProtocolInterfaces()	Yes	0
	InstallProtocolInterface()	Yes	552
	LocateDevicePath()	Yes	646
	LocateHandle()	Yes	0
	LocateHandleBuffer()	Yes	0
	LocateProtocol()	Yes	53,480
	OpenProtocol()	Yes	54,803
	OpenProtocolInformation()	Yes	810
	ProtocolsPerHandle()	Yes	373
	RegisterProtocolNotify()	Yes	65
	ReinstallProtocolInterface()	Yes	133
	UninstallMultipleProtocolInterfaces()	Yes	0
	UninstallProtocolInterface()	Yes	10
Task Priority	RaiseTPL()	Yes	1,181,652
	RestoreTPL()	Yes	1,181,524
Timer	GetNextMonotonicCount()	No (Depends on Monotonic Arch Protocol)	Not Measured
	SetWatchdogTimer()	No (Depends on Watchdog Arch Protocol)	5
	Stall()	No (Depends on Metronome Arch Protocol)	502

Patina Component Model

- Patina breaks away from individual PE32/COFF UEFI binaries to represent separate drivers to **a monolithic Patina DXE Core binary** that is built for a specific platform
  - Enables cross-module optimization, complete static analysis of control flow and memory ownership and lifetimes

- The platform creates a binary crate (using library crates from the patina repo) to produce a single “Patina DXE Core” UEFI binary that is placed in the platform firmware flash image

- The platform includes “components” in the binary

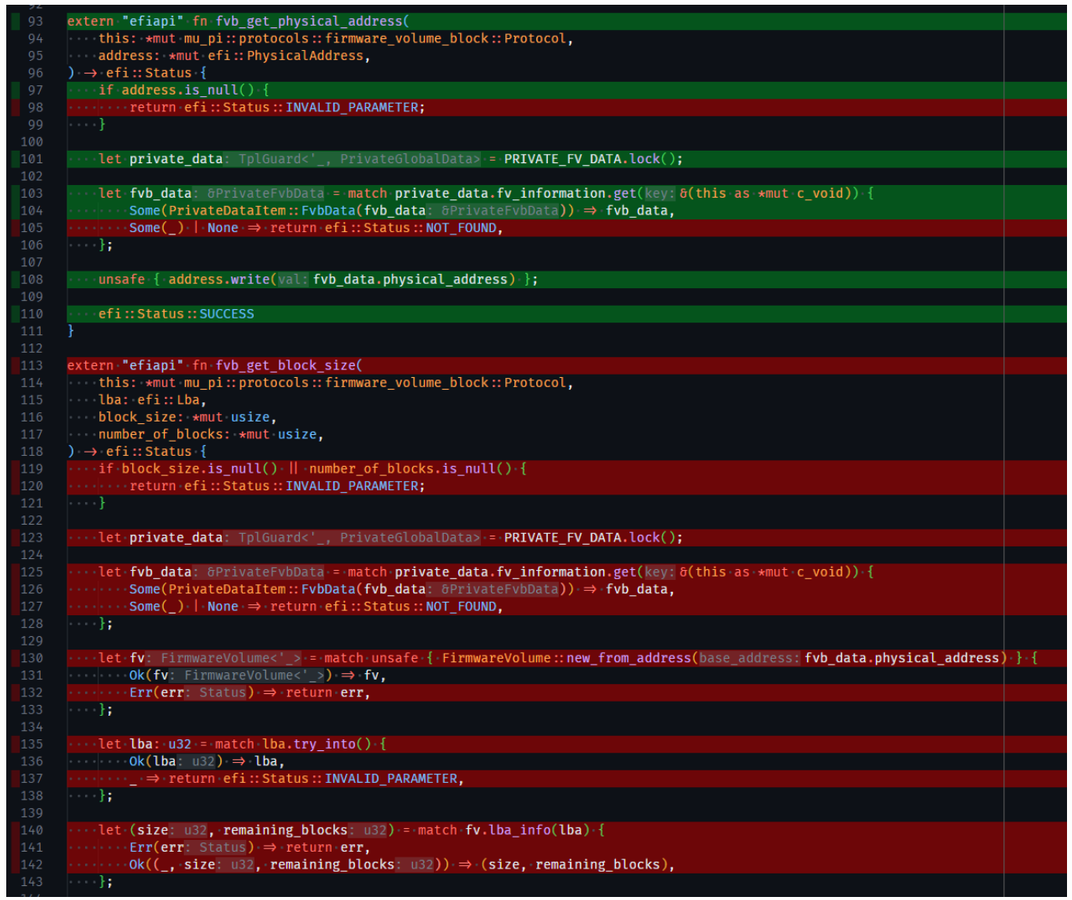
```rust
#[cfg_attr(target_os = "uefi", unsafe(export_name = "efi_main"))]
pub extern "efiapi" fn _start(physical_hob_list: *const c_void) -> ! {
    log::set_logger(&LOGGER).map(|()| log::set_max_level(log::LevelFilter::Trace)).unwrap();
    let adv_logger_component = AdvancedLoggerComponent::<Uart16550>::new(&LOGGER);
    adv_logger_component.init_advanced_logger(physical_hob_list).unwrap();

    patina_debugger::set_debugger(&DEBUGGER);

    log::info!("DXE Core Platform Binary v{}", env!("CARGO_PKG_VERSION"));

    Core::default()
        .with_section_extractor(patina_section_extractor::CompositeSectionExtractor::default())
        .init_memory(physical_hob_list) // We can make allocations now!
        .with_config(sc::Name("World")) // Config knob for sc::log_hello
        .with_component(adv_logger_component)
        .with_component(sc::log_hello) // Example of a function component
        .with_component(sc::HelloStruct("World")) // Example of a struct component
        .with_component(sc::GreetingsEnum::Hello("World")) // Example of a struct component (enum)
        .with_component(sc::GreetingsEnum::Goodbye("World")) // Example of a struct component (enum)
        .with_config(patina_mm::config::MmCommunicationConfiguration {
            acpi_base: patina_mm::config::AcpiBase::Mmio(0x0), // Actual ACPI base address will be set during boot
            cmd_port: patina_mm::config::MmiPort::Smi(0xB2),
            data_port: patina_mm::config::MmiPort::Smi(0xB3),
            comm_buffers: vec![],
        })
        .with_component(q35_services::mm_config_provider::MmConfigurationProvider)
        .with_component(q35_services::mm_control::QemuQ35PlatformMmControl::new())
        .with_component(patina_mm::component::sw_mmi_manager::SwMmiManager::new())
        .with_component(patina_mm::component::communicator::MmCommunicator::new())
        .with_component(q35_services::mm_test::QemuQ35MmTest::new())
        .with_config(patina_performance::config::PerfConfig {
            enable_component: true,
            enabled_measurements: {
                patina_sdk::performance::Measurement::DriverBindingStart         // Adds driver binding start measurements.
                | patina_sdk::performance::Measurement::DriverBindingStop        // Adds driver binding stop measurements.
                | patina_sdk::performance::Measurement::DriverBindingSupport     // Adds driver binding support measurements.
                | patina_sdk::performance::Measurement::LoadImage                // Adds load image measurements.
                | patina_sdk::performance::Measurement::StartImage // Adds start image measurements.
            },
        })
        .with_component(patina_performance::component::performance_config_provider::PerformanceConfigurationProvider)
        .with_component(patina_performance::component::performance::Performance)
        .start()
        .unwrap();

    log::info!("Dead Loop Time");
    loop {}
}
```

Patina Dev Processes

- Major changes requested w/ Requests for Comments (RFCs) through pull requests
- Track work using GitHub projects ("Patina project") and issues
- Publish documentation in an mdbook
- Publish using cargo-release
- Get unit test coverage using cargo-llvm-cov
- Check for undefined behavior using Miri
- Use cargo-deny in "cargo make all" command (local & continuous integration) to:
  - Check allowed licenses in dependencies (Apache-2.0, BSD-2-Clause-Patent, MIT)
  - Check crate dependencies against Rust security advisories (rustsec.org)
  - Check for redundant crates in the crate graph
  - Prevent crates being used in our deny list
- Use cargo-vet
  - OpenDevicePartnership crate audits are in a common repo - OpenDevicePartnership/rust-crate-audits

Unit Test Ergonomics

Up-to-date line coverage in VS Code:



```
93  extern "efiapi" fn fvb_get_physical_address(
94      this: *mut mu_pi::protocols::firmware_volume_block::Protocol,
95      address: *mut efi::PhysicalAddress,
96  ) -> efi::Status {
97      if address.is_null() {
98          return efi::Status::INVALID_PARAMETER;
99      }
100
101     let private_data: TplGuard<'_, PrivateGlobalData> = PRIVATE_FV_DATA.lock();
102
103     let fvb_data: &PrivateFvbData = match private_data.fv_information.get(key: &(this as *mut c_void)) {
104         Some(PrivateDataItem::FvbData(fvb_data: &PrivateFvbData)) => fvb_data,
105         Some(_) | None => return efi::Status::NOT_FOUND,
106     };
107
108     unsafe { address.write(val: fvb_data.physical_address) };
109
110     efi::Status::SUCCESS
111 }
112
113 extern "efiapi" fn fvb_get_block_size(
114     this: *mut mu_pi::protocols::firmware_volume_block::Protocol,
115     lba: efi::Lba,
116     block_size: *mut usize,
117     number_of_blocks: *mut usize,
118 ) -> efi::Status {
119     if block_size.is_null() || number_of_blocks.is_null() {
120         return efi::Status::INVALID_PARAMETER;
121     }
122
123     let private_data: TplGuard<'_, PrivateGlobalData> = PRIVATE_FV_DATA.lock();
124
125     let fvb_data: &PrivateFvbData = match private_data.fv_information.get(key: &(this as *mut c_void)) {
126         Some(PrivateDataItem::FvbData(fvb_data: &PrivateFvbData)) => fvb_data,
127         Some(_) | None => return efi::Status::NOT_FOUND,
128     };
129
130     let fv: FirmwareVolume<'_> = match unsafe { FirmwareVolume::new_from_address(base_address: fvb_data.physical_address) } {
131         Ok(fv: FirmwareVolume<'_>) => fv,
132         Err(err: Status) => return err,
133     };
134
135     let lba: u32 = match lba.try_into() {
136         Ok(lba: u32) => lba,
137         _ => return efi::Status::INVALID_PARAMETER,
138     };
139
140     let (size: u32, remaining_blocks: u32) = match fv.lba_info(lba) {
141         Err(err: Status) => return err,
142         Ok((_, size: u32, remaining_blocks: u32)) => (size, remaining_blocks),
143     };
```

File percentage coverage in VS Code:

HTML report coverage:

Back C:/\src/uefi-dxe-core/crates Covered: 964 of 1094 (88.12%)

Path	Coverage
uefi_collections	726 / 822 (88.32%)
uefi_component_interface	0 / 0
uefi_depex	107 / 135 (79.26%)
uefi_device_path	73 / 76 (96.05%)
uefi_test_macro	58 / 61 (95.08%)

Using the Patina DXE Core

QEMU Q35 Patina DXE Core

```rust
#[cfg_attr(target_os = "uefi", unsafe(export_name = "efi_main"))]
pub extern "efiapi" fn _start(physical_hob_list: *const c_void) -> ! {
    log::set_logger(&LOGGER).map(|()| log::set_max_level(log::LevelFilter::Trace)).unwrap();
    let adv_logger_component = AdvancedLoggerComponent::<Uart16550>::new(&LOGGER);
    adv_logger_component.init_advanced_logger(physical_hob_list).unwrap();

    patina_debugger::set_debugger(&DEBUGGER);

    log::info!("DXE Core Platform Binary v{}", env!("CARGO_PKG_VERSION"));

    Core::default()
        .with_section_extractor(patina_section_extractor::CompositeSectionExtractor::default())
        .init_memory(physical_hob_list) // We can make allocations now!
        .with_config(sc::Name("World")) // Config knob for sc::log_hello
        .with_component(adv_logger_component)
        .with_component(sc::log_hello) // Example of a function component
        .with_component(sc::HelloStruct("World")) // Example of a struct component
        .with_component(sc::GreetingsEnum::Hello("World")) // Example of a struct component (enum)
        .with_component(sc::GreetingsEnum::Goodbye("World")) // Example of a struct component (enum)
        .with_config(patina_mm::config::MmCommunicationConfiguration {
            acpi_base: patina_mm::config::AcpiBase::Mmio(0x0), // Actual ACPI base address will be set during boot
            cmd_port: patina_mm::config::MmiPort::Smi(0xB2),
            data_port: patina_mm::config::MmiPort::Smi(0xB3),
            comm_buffers: vec![],
        })
        .with_component(q35_services::mm_config_provider::MmConfigurationProvider)
        .with_component(q35_services::mm_control::QemuQ35PlatformMmControl::new())
        .with_component(patina_mm::component::sw_mmi_manager::SwMmiManager::new())
        .with_component(patina_mm::component::communicator::MmCommunicator::new())
        .with_component(q35_services::mm_test::QemuQ35MmTest::new())
        .with_config(patina_performance::config::PerfConfig {
            enable_component: true,
            enabled_measurements: {
                patina_sdk::performance::Measurement::DriverBindingStart         // Adds driver binding start measu
                | patina_sdk::performance::Measurement::DriverBindingStop        // Adds driver binding stop measure
                | patina_sdk::performance::Measurement::DriverBindingSupport     // Adds driver binding support meas
                | patina_sdk::performance::Measurement::LoadImage                // Adds load image measurements.
                | patina_sdk::performance::Measurement::StartImage // Adds start image measurements.
            },
        })
        .with_component(patina_performance::component::performance_config_provider::PerformanceConfigurationProvide
        .with_component(patina_performance::component::performance::Performance)
        .start()
        .unwrap();

    log::info!("Dead Loop Time");
    loop {}
}
```

```rust
static DEBUGGER: patina_debugger::PatinaDebugger<Uart16550> =
    patina_debugger::PatinaDebugger::new(Uart16550::Io { base: 0x3F8 })
        .with_force_enable(false)
        .with_log_policy(patina_debugger::DebuggerLoggingPolicy::FullLogging);
```

```rust
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    log::error!("{}", info);

    if let Err(err) = unsafe { StackTrace::dump() } {
        log::error!("StackTrace: {}", err);
    }

    if patina_debugger::enabled() {
        patina_debugger::breakpoint();
    }

    loop {}
}
```

Patina FW Patching

[OpenDevicePartnership/patina-fw-patcher: A tool to quickly patch Patina FW into a UEFI ROM.]

We can build the Patina DXE Core EFI binary, patch it into an existing UEFI FW image, and run that image in QEMU within seconds. This is the recommended dev and test path.

Next Steps

- Continue minimizing the use of unsafe code in Patina
- Write more Patina components
- Add support for a Management Mode (MM) core
- Updating Specs: The UEFI Specification, Platform Initialization (PI) Specification, and other specifications have interfaces with C in mind
- Collaborate on Patina with the community. Some topic areas:
  - Rust testing techniques and frameworks
  - Crate organization and code layout
  - The Patina component model
  - The Patina build process
  - Opportunities to share code
  - Patina platform integration and the Patina Readiness Tool
  - Any questions you might have about Rust and Rust projects you may be working on

Questions?

References

- Data Sources
  - DHS CISA Strategy to Fix Vulnerabilities Below the OS Among Worst Offenders
  - UEFI 2020 Virtual Plugfest - Enabling RUST for UEFI Firmware
- Firmware Projects
  - tianocore/edk2: EDK II
  - tianocore/edk2-staging at edkii-rust
- Organizations and Specifications
  - Microsoft Enhanced Memory Protections
  - Open Device Partnership
  - Rust Documentation
  - Rust Programming Language
  - Specifications | Unified Extensible Firmware Interface Forum
- Rust Projects
  - r-efi/r-efi: UEFI Reference Specification Protocol Constants and Definitions
  - sagiegurari/cargo-make: Rust task runner and build tool.
  - taiki-e/cargo-llvm-cov: Cargo subcommand to easily use LLVM source-based code coverage (-C instrument-coverage).
  - crate-ci/cargo-release: Cargo subcommand `release`: everything about releasing a rust crate.
  - rust-lang/miri: An interpreter for Rust's mid-level intermediate representation
  - EmbarkStudios/cargo-deny: ❌ Cargo plugin for linting your dependencies 🦀
  - mozilla/cargo-vet: supply-chain security for Rust