CPP-Summit 2019

全球C++软件技术大会

C++ Development Technology Summit

Boolan

高端IT互联网教育平台

关注“博览Boolan”服务号

发现更多 会议·课程·活动

If You Can't Open It You Don't Own It

MATTHEW BUTLER

CPP-SUMMIT • NOVEMBER 2, 2019

About Me

- Software architect and security researcher
  - Started with C++ professionally in 1990 (Borland C++ 1.0)
  - C++Now and CppCon staff
  - International conference speaker and trainer

- Areas of expertise:
  - Network and applications security
  - Safety critical systems
  - Real-time data analysis
  - Embedded systems

- Member of the ISO C++ Standards Committee
  - Evolution Working Group (EWG)
  - SG12 – Software vulnerabilities and safety critical systems
  - SG14 – Low Latency, embedded
  - SG21 – Contracts

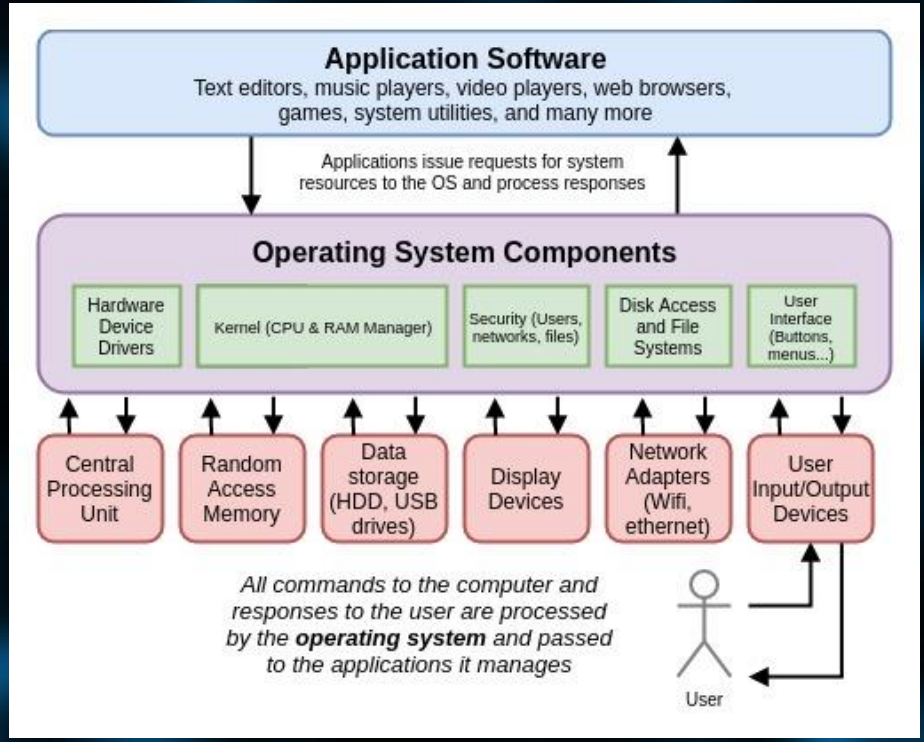
ShadowHammer (2018)

- Highly complex and sophisticated supply chain attack
  - Threat actors compromised ASUS's automatic update servers
  - Injected malicious code into firmware updates
  - The infected malware was signed by ASUS
  - ASUS pushed malware to ~500K computers
- Malware reached out the C^2 servers
  - Opened backdoors to the infected PCs
  - Installed more malware and began moving laterally within their systems
- Targeted ~600 computers by MAC Address

Roots Of Trust

Hardware Roots of Trust

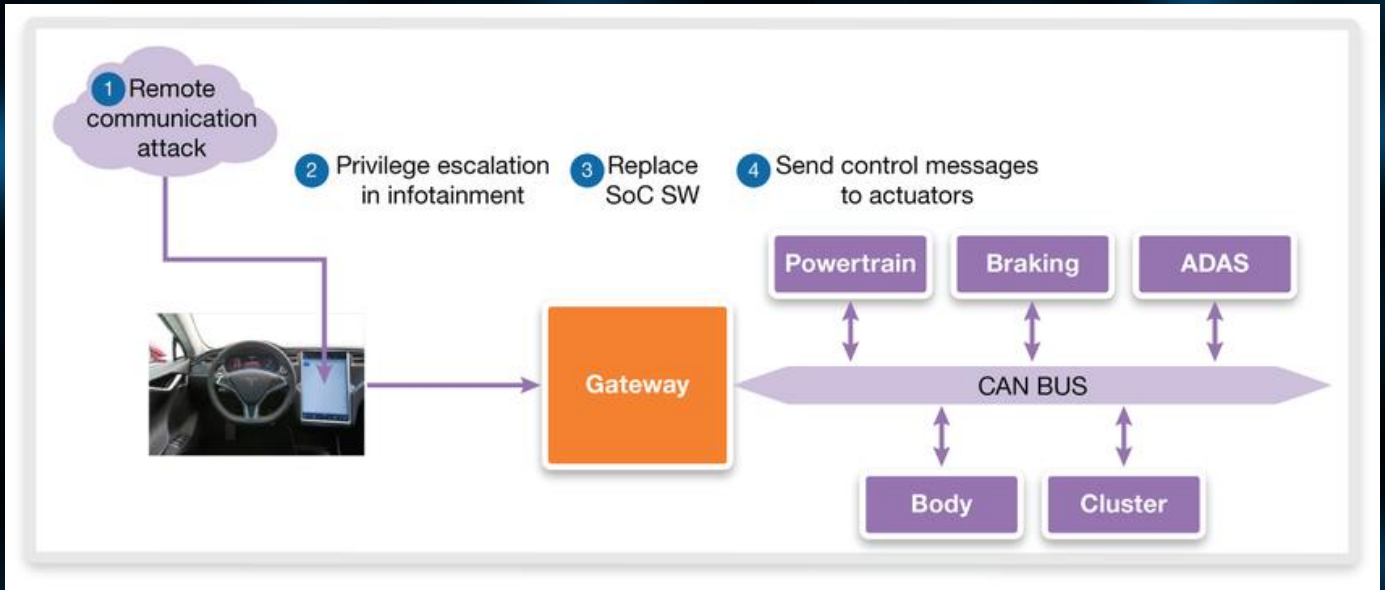
- Trust landscape
  - Each layer implicitly trusts the layer it's built upon
  - The lower the level the harder the trust is to break
  - Trust is organic
- Hardware has the highest trust level
  - It has the smallest attack surface
  - It's farthest from the attack points
  - It's harder to understand
  - Attack vectors are harder to realize



Source: Technology Rediscovery

Hardware Roots of Trust

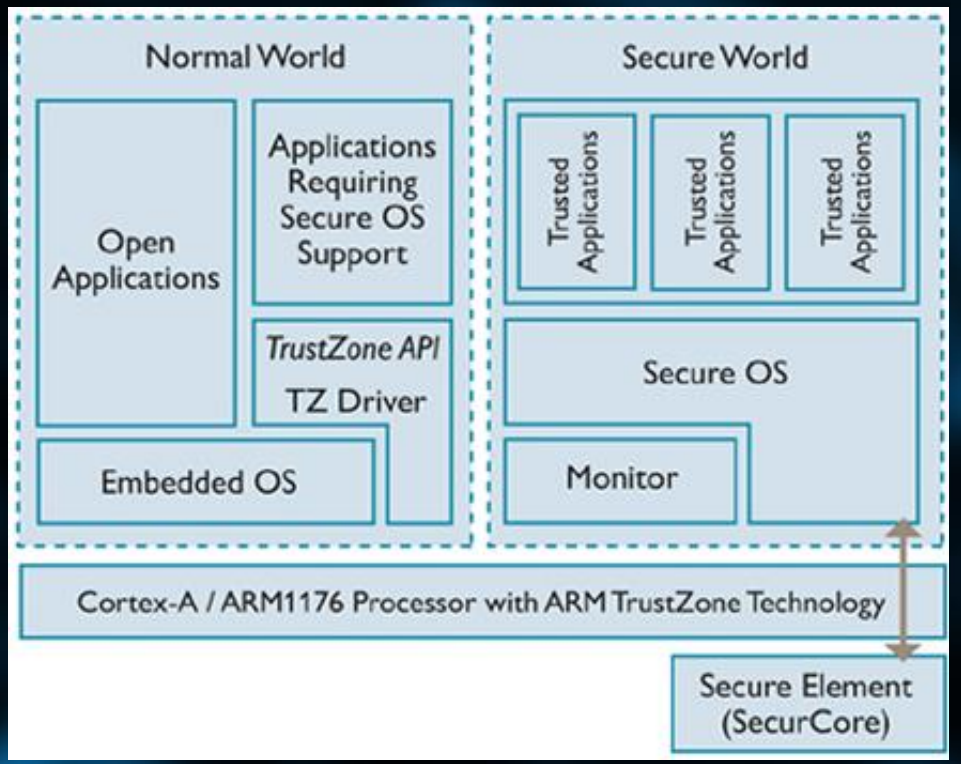
- Broken trust
  - What happens when that trust is broken?
  - Undermining the lower layers undermines the layers above
  - Allows lateral movement within a system
  - Trust is organic
- Physical security
  - Shared busses
  - Mixed trusts
  - Poor comm security



Source: Synopsys

Hardware Roots of Trust

- Trusted Execution Environment (TEE)
  - Secure area inside the processor
  - Isolated environment
  - Cryptographic Services
  - Anti-Tamper Services
  - Trusted Device Identity
- Microsoft's Secure-Core
  - Firmware has a higher privilege than the kernel
  - Apple controls their hardware

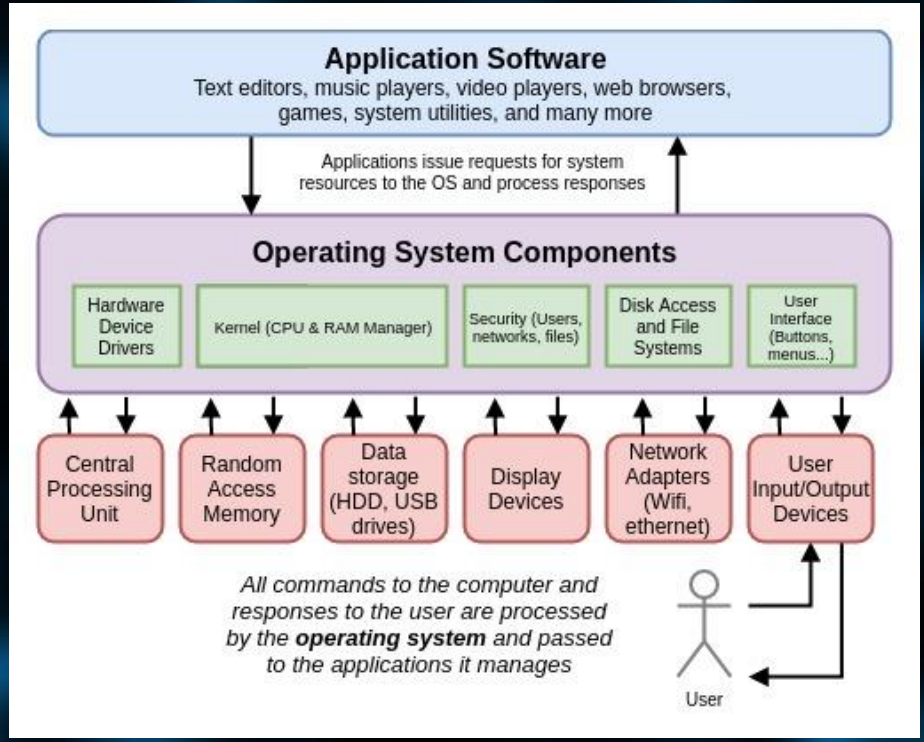


Source: Synopsys

Software Roots of Trust

- Our software explicitly trusts the layers it rests on
  - Kernel
  - System libraries
  - Hardware
  - Firmware

- Implicitly trusts system constraints
  - Permissions
  - File system ACLs
  - CPU architecture
  - Communications pathways
  - Update system



Source: Technology Rediscovery

Software Roots of Trust

- Mostly bad news for what you can trust
  - You have to trust the HW…but you don't own it
  - You have to trust the kernel…but you don't control it
  - You have to trust the system constraints…but they can be by-passed
  - You have to trust the system libraries…but they can be replaced
  - You have to trust the system updates…but they can be breached
  - You have to trust encrypted transport layers…but they can be broken
- So what can we trust?

Software Roots of Trust

- You trust what you control
  - What you compile
  - The libraries you use
  - Who and what you connect with
  - What you hold in memory and how long you hold it

Software Roots of Trust

- What you compile
  - Don't mix trust levels in code
  - Sign your code and verify the signature on start-up
  - Encrypt your traffic (even if the transport layer is already encrypted)

- The libraries you use
  - Don't trust open source unless a) you wrote it or b) you code reviewed it close enough to re-write it or c) you know (and trust) the person who owns it
  - Know the pedigree of your libraries
  - Library Poisoning is still a thing
  - We'll talk more about Supply Chains

Software Roots of Trust

- Who and what you connect with
  - Don't rely on the OS permissions and ACLs
  - Verify who you're doing business with
  - Force authentications on internal and external communications

- What you hold in memory and how long you hold it
  - Don't burn your keys into the code
  - Shared secrets are not really secret
  - Use TPM (Trusted Platform Module) or Secure Stores
  - Only hold keys while they're in use
  - Clearing memory doesn't always clear it

Take-aways

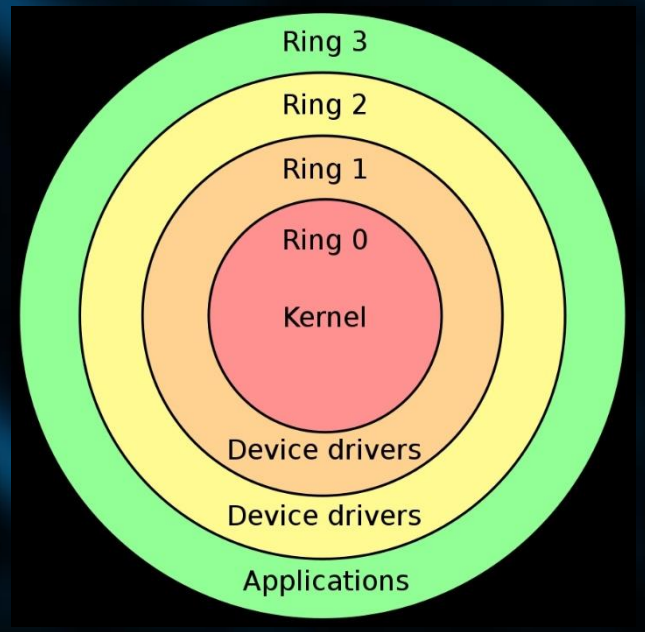
- Trusted systems should have no connection to untrusted systems
- System internals are a Zero Trust environment
- All memory is eidetic
- Sign your code and validate your binaries
- Manage your own eco system
- Use secure storage for your keys
- “Trust But Verify”
- Ultimately you own almost nothing that you trust

x86 CPU God Mode (2018)

"Additionally, accessing some of the internal control registers can enable the user to bypass security mechanisms, e.g., allowing ring 0 access at ring 3. In addition, these control registers may reveal information that the processor designers wish to keep proprietary. For these reasons, the various x86 processor manufacturers have not publicly documented any description of the address or function of some control MSRs."

Patent #US8341419

x86 CPU God Mode (2018)



-1: The Hypervisor

-2: System Management Code

-3: Q35 / AMT / ME

MSRs are Model-Specific Registers

64-bit Control Registers

Accessed by address not name

Use ***rdmsr*** and ***wrmsr*** instructions

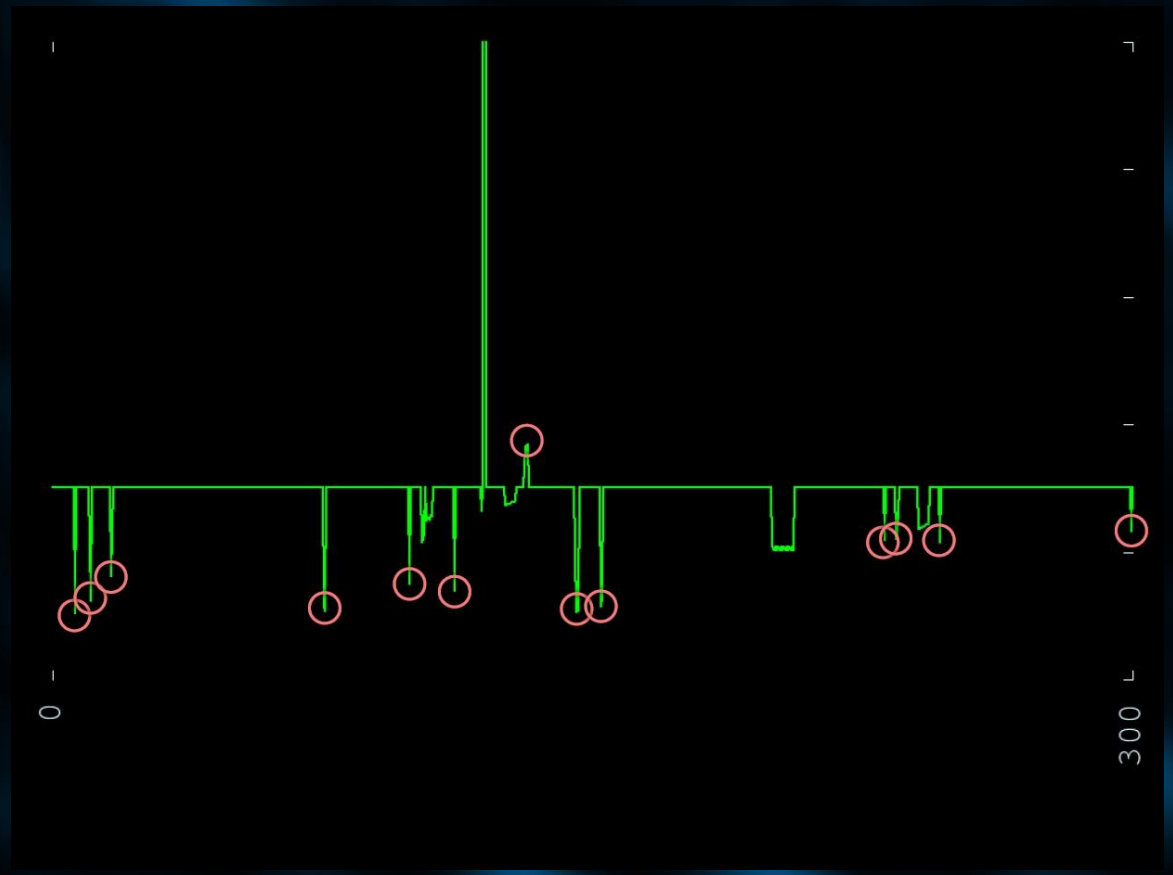
Only accessible from ring 0

So how do we find out which MSRs exist?

Try them all and the ones that don't fault, exist.

This led to ***1300*** MSRs…

x86 CPU God Mode (2018)



Source: Christopher Domas

Calculate the access time for each MSR

Functionally ***different*** MSRs should have different access times

Functionally ***equivalent*** MSRs should have approximately the same access times

Side channel (timing) attack differentiates between MSRs

Identified ***43*** unique MSRs from the ***1300*** possible MSRs

Reduces the problem to 2752 bits to check

Still leaves ~1.3×10^{36} x86 instructions

Which is ~eternity

Enter ***Sandsifter*** which fuzzes the MSRs

x86 CPU God Mode (2018)

Source: Christopher Domas

Sandsifter scans the x86 ISA in about a day

Found the launch instruction ***0x0F3F*** which is essentially ***jmp %eax*** (a launch instruction)

Next, test bits one by one

Found ***MSR 1107, bit 0*** is the God Mode…

…in about a week!

This bypasses all of the security checks…

Side Channels

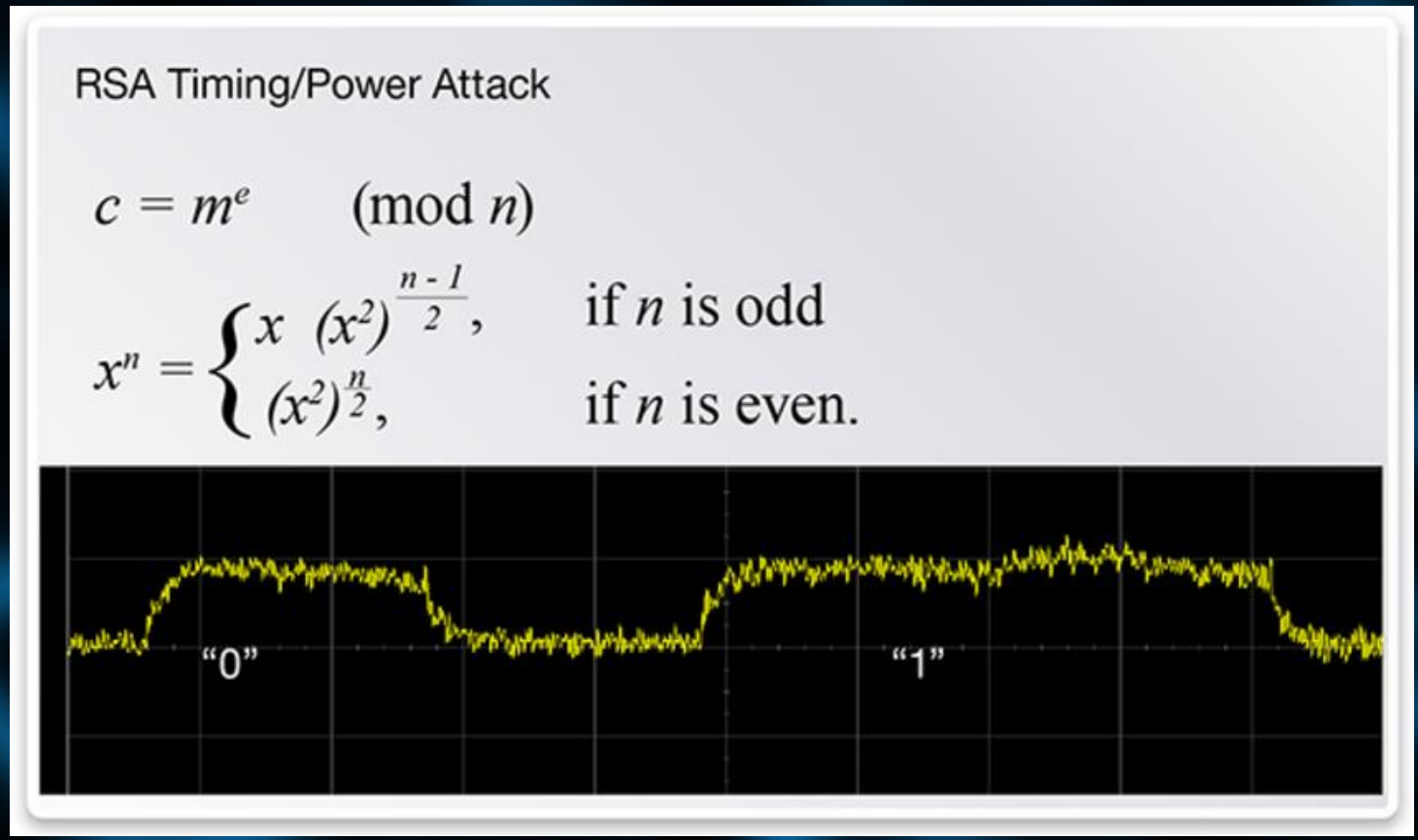
Hardware Side Channels

- Timing
  - Running time is proportional to the ***number*** of circuits
  - We can deduce the timing characteristics of each circuit
  - We can detect what's operational at any given time

- Power
  - Power is proportional to the ***number*** and ***type*** of circuits
  - We can deduce how much current each circuit draws
  - We can detect what's operational at any given time

Hardware Side Channels

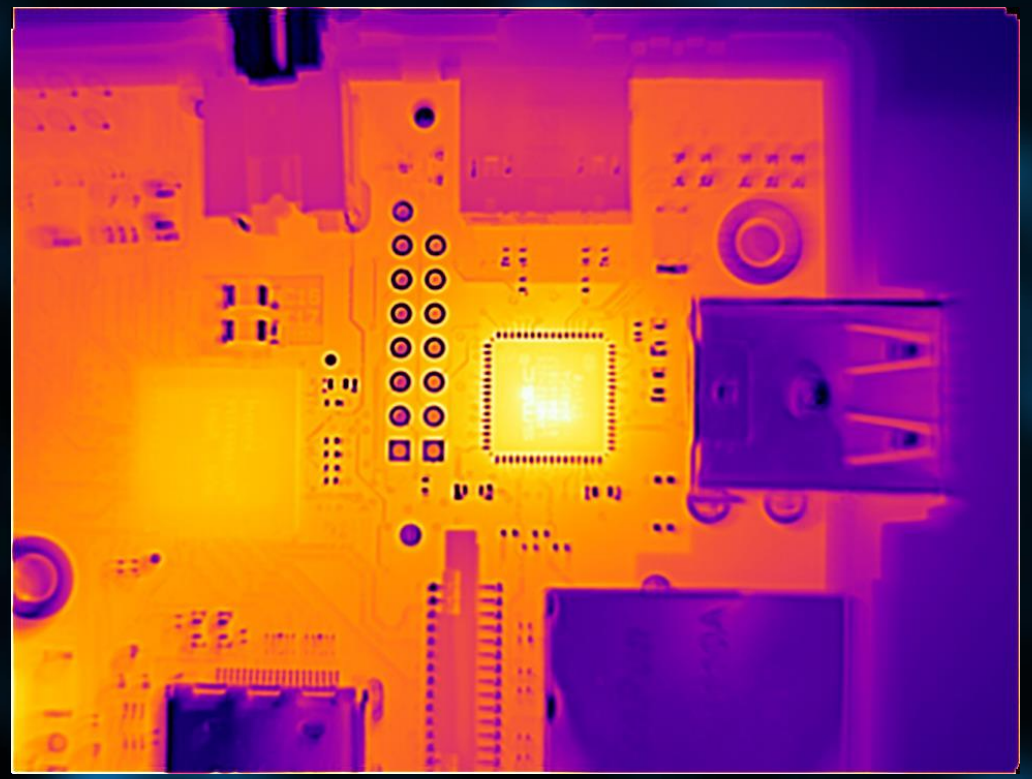
- Differential Power Analysis



Source: Synopsys

Hardware Side Channels

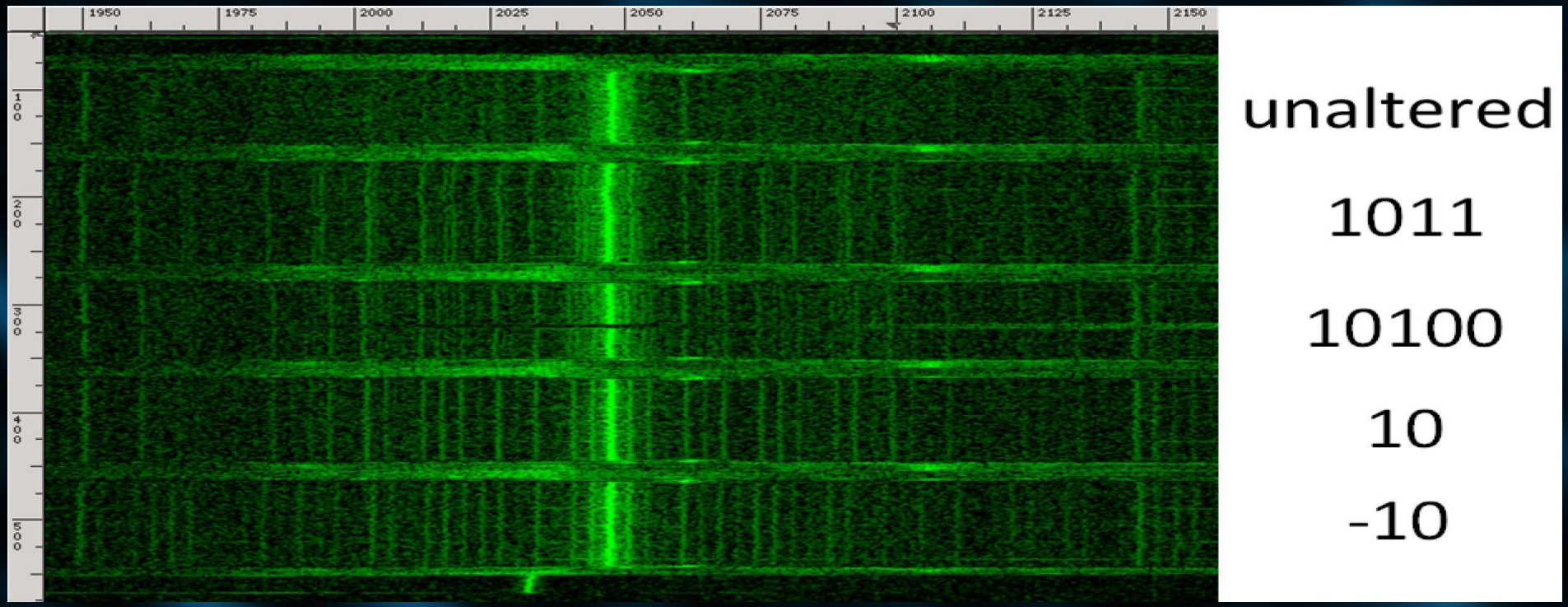
- Differential Fault Analysis
  - Inject faults into a system
  - Measure the outcome
- Emissions (TEMPEST)
  - RF emissions
  - Thermal signatures
  - Acoustic noise



Source: Mike Anderson, PTR

Hardware Side Channels

- TEMPEST (Side Band Analysis)



Source: Daniel Genkin, et al

Software Side Channels

- Timing
  - Cached data – CacheBleed (SSL), Spectre and Meltdown
  - Concurrency
  - Response timing

- Memory
  - Forced memory overflows that alter control flow
  - Holding keys in memory after use
  - Data type conversions – undefined behavior

Software Side Channels

- Wi-fi
  - Wi-Fi traffic is easily "sniffed' using a promiscuous mode NIC
  - Key Reinstallation Attacks (KRACKs) have compromised WPA2
  - WPA has been broken
- Network traffic
  - Even encrypted traffic is vulnerable to statistical analysis – See Netflix
  - Response sizes & timings are data

Software Side Channels

- Logging
  - Accounts or account information
  - Verbose logging
  - Function names and parameters
  - Configuration information
  - Hardware setup information
- Error messages
  - Paths and locations of sensitive files
  - Accounts or account information
  - Configuration information

Software Side Channels

- Licensing that requires source code distribution
  - Requires that you distribute any modified source code
  - Licenses: GNU, Mozilla
  - Examples: Busy Box, QT
- Licensing that requires attribution
  - Require you to disclose use of the library in your product
  - Licenses: GNU, Mozilla, Apache, MIT
  - Examples: Busy Box, OpenSSL 3.x, BDE

Take-aways

- Side Channel attacks can target anything you "leak"
- Understand the effects of open source licensing
- Sensor logs and error messages
- Turn down logging levels
- Limit external emissions of information
- Use constant sized responses and timings
- Don't rely on first line crypto
- Limit the amount of time sensitive data stay in memory

Diginotar (2011)

- Dutch Certificate Authority
  - Threat actors generated more that 500 fake certificates
  - Including a wildcard certificate for Google
  - Targeted 300,000 Iranian Gmail users
  - Ran man-in-the-middle attacks against Google services
  - The hacker left a massage: *"I Will sacrifice my soul for my leader"* written in Persian
- Diginotar had exceptional security
  - Layered network security
  - Physical security on the certificate server
- Sort of…
  - Penetration came though an external, unpatched website
  - Physical security was bypassed by leaving a key card in reader permanently

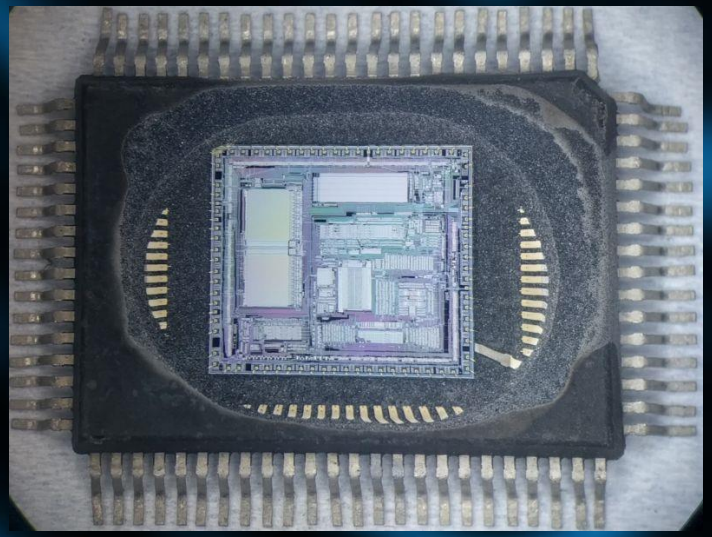
Physical Access

Hardware Reverse Engineering

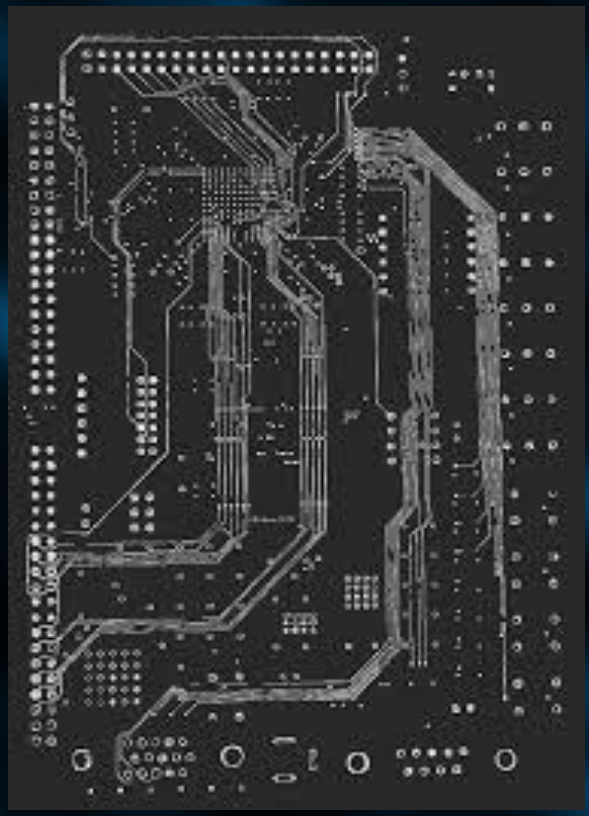
- Non-Invasive
  - Remote – Communications protocols
  - Local – Debug access
- Invasive
  - Physically Invasive – Fault injection
  - Destructive – Decapsulation

Hardware Reverse Engineering

- PCB Reverse Engineering
  - Datasheets
  - X-ray Computed Tomography
  - Decapsulation



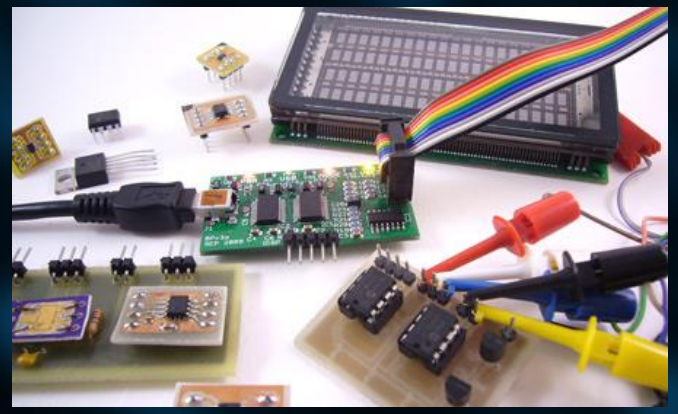
Source: Ars Technica



Source: Navid Asadizanjani, et al

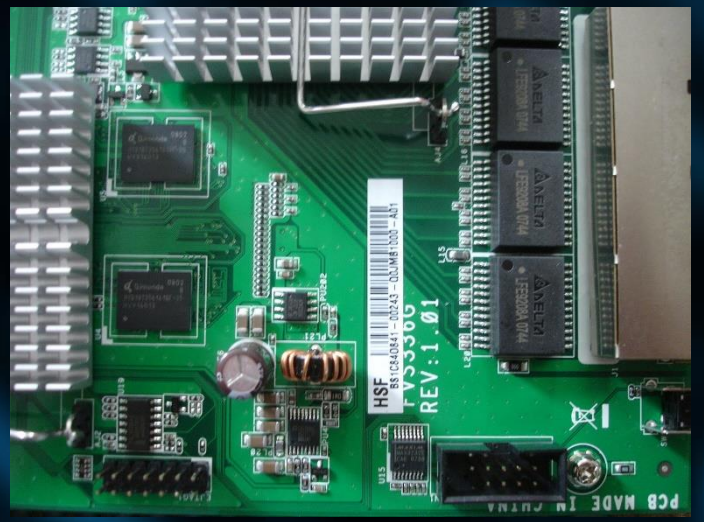
Hardware Reverse Engineering

- Interface Analysis
  - Bus Pirate – Signal tester
  - Digital Logic Analyzer – Protocol analysis
  - SPI, I2C, CAN, RS-252, TCP



Source: Bus Pirate

- Connector Repopulation
  - JTAG / Single Wire Debug – Read firmware
  - Serial Ports – Watch boot cycle



Source: Netgear

Software Reverse Engineering

- Vendor Information
  - Website
  - Open FTP sites
  - Dark Web
- Log files
  - We're really chatty in our logs
  - Configuration information, control flow, function names, et al
- System Artifacts
  - Config files
  - Databases
  - Registry

Software Reverse Engineering

- Binaries
  - Did you ship debug code?
  - Did you strip out the symbols?
  - Did you bake crypto keys into the binary?

  - Requires a high-level of assembly knowledge

- Tools
  - strace, ltrace, drtrace – Debuggers
  - IDA Pro – Disassembler / Debugger
  - Virtuailor (Gal Zaban) – Reconstructs C++ virtual tables

Software Reverse Engineering

- IPC Interfaces
  - Rarely secured
  - Very useful for "fuzzing" attacks
  - Encrypt your traffic and authentic traffic
  - Session keys help eliminate replay and man-in-the-middle attacks

- Protocols
  - Control flow languages are generally sent in the clear
  - These allow attackers to understand sources and methods
  - Treat protocols like IP

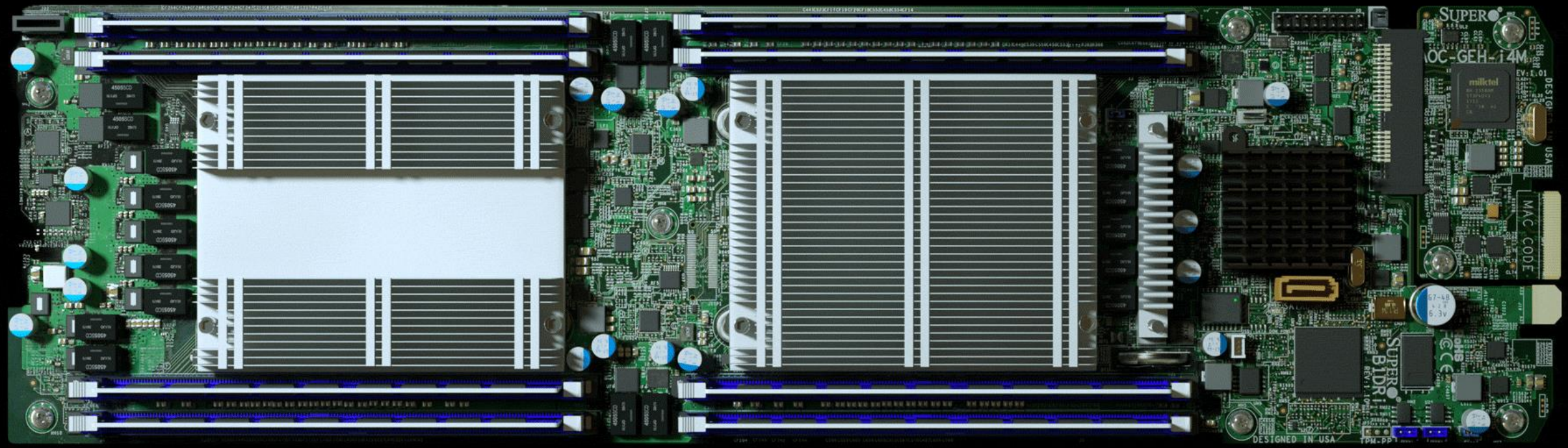
Take-aways

- Always secure your communications paths
- Obfuscation is almost always a bad strategy
- Never ship CLIs for your support engineers
- Strictly limit what goes into your logs
- Eliminate the use of the Windows registry
- Encrypt your config files
- Don’t bake your crypto keys into your binaries
- Use hardware crypto (TPM) & key stores
- Treat control flow languages as IP

Supermicro Supply Chain Hack (2015)

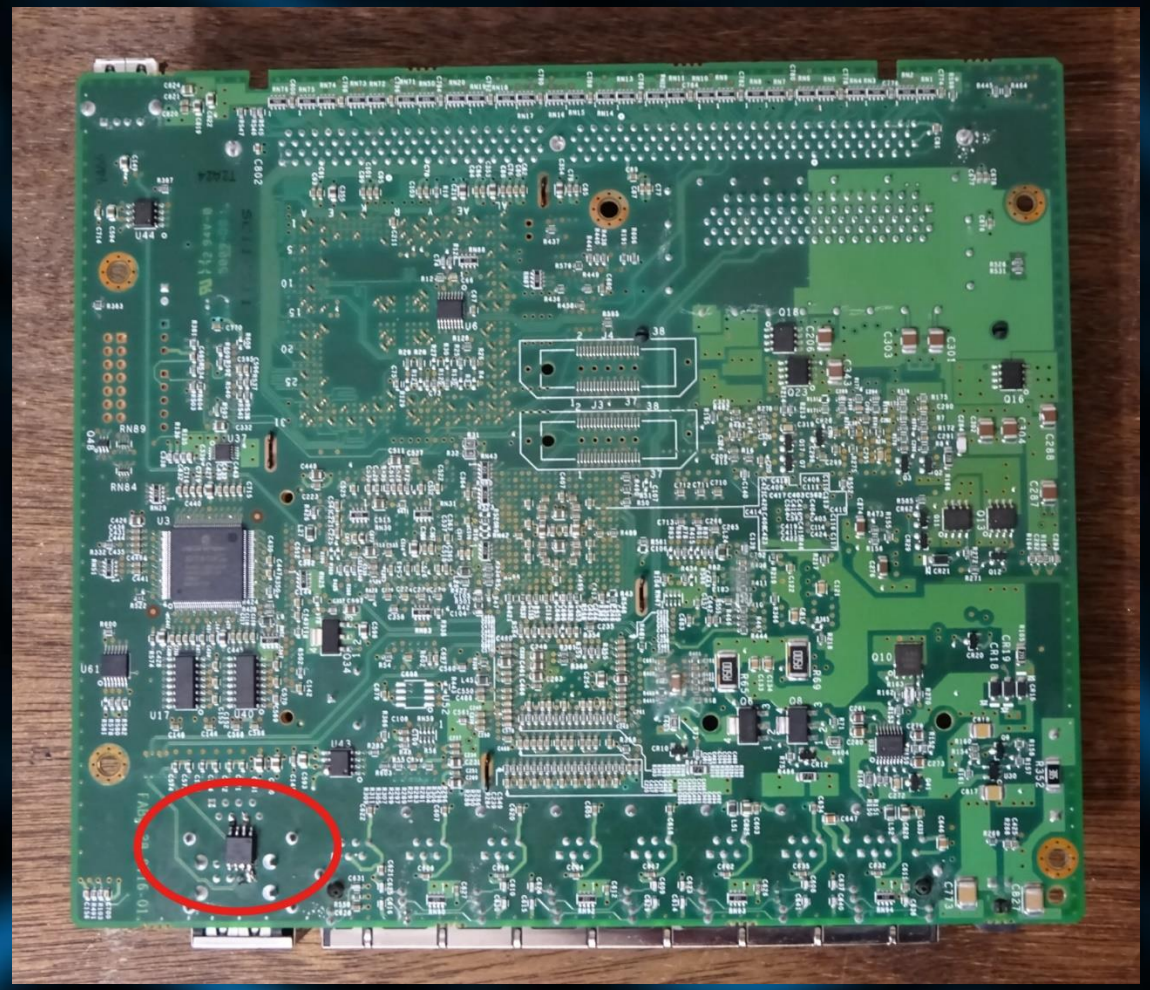
- Nation state level hack
  - Nestled in the servers' motherboards was a microchip
  - Microchip was the size of a grain of rice
  - Allowed the threat actor to create a stealth doorway into the network
  - Targets: Elemental, DoD, CIA, Apple and 30 others
  - Some elements of the story did not ring completely true
- Fallout
  - Most companies straight up denied the Bloomberg report
  - Bloomberg stood by their reporting

Supermicro Supply Chain Attack (2015)



Source: Bloomberg Businessweek

Hardware



- Security researcher Monta Elkins
  - ATtiny85 chip (5mm x 5mm)
  - Cisco ASA 5505 firewall
  - $200 in parts and equipment
- Bypasses the security system
  - Carries out an attack on boot
  - Impersonates a security administrator
  - Triggers the firewall's password recovery feature
  - Creates a new admin account

Supply Chains

Hardware Supply Chain Hacks

- Interdiction
  - Shipments are intercepted and hardware modified
  - Eliminates the need to compromise the manufacturer
  - Requires a high level of technical capability
  - Very difficult technically to pull off
- Seeding
  - Malicious hardware is added at the point of manufacturing
  - Requires compromising the manufacturer
  - Requires a nation state level of capability
  - Easier technically to pull off

Hardware Supply Chain Hacks

- Random testing
  - Pull random units in for testing of hardware and firmware
  - Destructive and non-destructive testing
- Onsite inspections
  - Surprise on-site inspections are part of every contract (or not)
  - Verify hardware configurations
- Monitoring for anomalous behavior
  - Monitor for spurious signals and traffic
  - Look at power utilization curves

Hardware Supply Chain Hacks

- Random testing
  - I don't need to compromise every unit
  - Odds of finding the one compromised unit in a batch are low
- Onsite inspections
  - Anyone smart enough to do this is clever enough not to get caught
  - These are complex processes
- Monitoring for anomalous behavior
  - Non-deterministic and defeated by beaconing
  - What do you look for?

Event-Stream (2018)

- Node.js NPM package
  - 2M downloads per week
  - Users can download source code or binaries
- Original author "transferred" control to a new maintainer: ***right9control***
  - No vetting of any kind (and no warning)
  - right9control added a dependency: Flatmap-Stream
  - Flatmap-Stream was modified to include bitcoin stealing code
  - Steals Bitcoin and Bitcoin Cash funds stored inside BitPay's Copay wallet apps
- If you used this library, you were a bitcoin miner
  - Reviewing the source did not help
  - Only the binaries had the malicious code

Software Supply Chains

- Open Source
  - Have you code reviewed all the open source libraries you use?
  - Do you trust the authors of the open source libraries you use?
  - Do you even know the authors of the open source libraries you use?
  - Have you code reviewed the library for security?

- Closed Source
  - Is the company reputable?
  - Have their libraries been code reviewed for security?

Software Supply Chains

- Code review your third-party libraries
  - Don't assume that they're safe (or bug free) – Open SSL
  - Prefer libraries that have been security reviewed
  - Never use pre-compiled libraries
  - Invest in code scanning products
  - Submit bug reports off channel directly to the authors
  - Practice tough love

Software Supply Chains

- Test your third-party libraries
  - Use the library in isolation
  - Look for spurious network traffic
  - Look for spurious HD traffic
  - Does it open ports?
  - Does it try to connect to other ports?

Take-aways

- "You're Only As Strong As Your Weakest Third-Party Library"
- Never assume that supply chains are secure
- Test your third-party libraries
- "Trust But Verify"
- Learn how to conduct security audits & code reviews
- Use products such as Black Duck & WhiteSource
- Practice tough love

C++20 And Security

Security In C++20

- Concepts
- Ranges
- std::atomic<std::shared_ptr<T>>
- Volatile deprecation
- Designated Initializers
- using enum
- std::format replaces printf()

Concepts

- Has been described as a “type-system for the type-system”
- Based on the C++14 Concepts TS
- Allows you to describe constraints in generic code that helps guide template instantiation

Concepts


```
1. template<typename T>
2. inline constexpr bool is_sortable = //...
3.
4. template<typename T>
5. void sort(T& t) {
6.    static_assert(is_sortable<T>, "Type T is not sortable");
7.    ...
8. }
```

```
1. template<bool B>
2. using requires_t = std::enable_if<B, bool>;
3.
4. template<typename T,
5.    requires_t<is_sortable<T>> = true>
6. void sort(T& t) {
7.         ...
8. }
```

Concepts


```
1. template<class I, class R = ranges::less, class P = identity>
2.    concept sortable =
3.      permutable<I> &&
4.      indirect_strict_weak_order<R, projected<I, P>>;
```

```
1. template <sortable T>
2. void sort(T& T) {
3.    ...
4. }
```

Concepts Already Defined

- movable
- copyable
- swappable
- sortable
- totally_ordered
- derived_from
- convertible_to
- assignable_from
- destructible
- default_constructible
- constructible_from
- move_constructible
- copy_constructible
- same_as
- common_reference_with
- common_with
- equality_comparable
- invocable
- regular_invocable
- integral
- signed_integral
- unsigned_integral
- floating_point
- boolean
- semiregular
- regular

Ranges


```
1.  std::vector<int> vec{ 10, 5, 2, 67, 12, 32, 19, 44, 12 };
2.  std::sort(vec.begin(), vec.end());
```

Ranges


```
1.  std::vector<int> vec{ 10, 5, 2, 67, 12, 32, 19, 44, 12 };
2.  std::sort(vec.begin(), vec.end());
```

```
1.  std::vector<int> vec{ 10, 5, 2, 67, 12, 32, 19, 44, 12 };
2.  ranges::sort(vec);
```

Ranges


```
1.  std::vector<int> vec{ 10, 5, 2, 67, 12, 32, 19, 44, 12 };
2.  std::sort(vec.begin(), vec.end());
```

```
1.  std::vector<int> vec{ 10, 5, 2, 67, 12, 32, 19, 44, 12 };
2.  ranges::sort(vec);
```

```
1.  std::map<string, int> mp;
2.  mp["foo"] = 12;
3.  mp["bar"] = 43;
4.
5.  for (auto [s, v] : ranges::reverse_view(mp)) {
6.     cout << s << : << v << "\n";
7.  }
```

```
1. std::vector<int> masks{ 10, 5, 2, 67, 12, 32, 19, 44, 12 };
2.
3. auto bit_1_mask = [](int val) { return 0x0010 = val & 0x0010; };
4.
5. auto bit_1_is_masked = masks | ranges::view::filter(bit_1_mask);
6.
7. for (int masked : bit_1_is_masked) {
8.    std::cout << masked << "\n";
9. }
```

std::atomic<std::shared_ptr<T>>

- Is std::shared_ptr<T> thread safe?

std::atomic<std::shared_ptr<T>>

- Is std::shared_ptr<T> thread safe?

- Yes and No
- The control block is
- But the T is not

std::atomic<std::shared_ptr<T>>


```
1.  template<typename T> class concurrent_stack {
2.     struct Node { T t; shared_ptr<Node> next; };
3.     shared_ptr<Node> head;
4.  public:
5.     class reference {
6.        shared_ptr<Node> p;
7.     public:
8.        reference(shared_ptr<Node> p_) : p{ p_ } { }
9.        T& operator*() { return p->t; }
10.       T* operator->() { return &p->t; }
11.    };
12.    auto find(T t) const {
13.       lock_guard<std::mutex> lock(m_mutex);
14.       auto p = atomic_load(&head);
15.       while (p && p->t != t)
16.          p = p->next;
17.       return reference(move(p));
18.    }
19.    void push_front(T t) {
20.       lock_guard<std::mutex> lock(m_mutex);
21.       auto p = make_shared<Node>();
22.       p->t = t;
23.       p->next = atomic_load(&head);
24.       while (!atomic_compare_exchange_weak(&head, &p->next, p)) {}
25.    }
26.    auto front() const {
27.       lock_guard<std::mutex> lock(m_mutex);
28.       return reference(atomic_load(&head));
29.    }
30.    void pop_front() {
31.       lock_guard<std::mutex> lock(m_mutex);
32.       auto p = atomic_load(&head);
33.       while (p && !atomic_compare_exchange_weak(&head, &p, p->next)) {}
34.    }
35. };
```

Source: Herb Sutter, Atomic Smart Pointers

std::atomic<std::shared_ptr<T>>


```
1.  template<typename T> class concurrent_stack {
2.     struct Node { T t; shared_ptr<Node> next; };
3.     atomic<shared_ptr<Node>> head;
4.  public:
5.     class reference {
6.        shared_ptr<Node> p;
7.     public:
8.        reference(shared_ptr<Node> p_) : p{ p_ } { }
9.        T& operator*() { return p->t; }
10.       T* operator->() { return &p->t; }
11.    };
12.    auto find(T t) const {
13.
14.       auto p = head.load();
15.       while (p && p->t != t)
16.          p = p->next;
17.       return reference(move(p));
18.    }
19.    void push_front(T t) {
20.
21.       auto p = make_shared<Node>();
22.       p->t = t;
23.       p->next = head;
24.       while (!head.compare_exchange_weak(p->next, p)) {}
25.    }
26.    auto front() const {
27.
28.       return reference(head.load());
29.    }
30.    void pop_front() {
31.
32.       auto p = head.load();
33.       while (p && !head.compare_exchange_weak(p, p->next)) {}
34.    }
35. };
```

Source: Herb Sutter, Atomic Smart Pointers

Volatile Deprecation

- Is designed to suppress optimizations where registers can change outside the view of the compiler
- We still need it, so it's not going away

- Because it's a regular CV-qualifier it tends to get used in some sketchy ways
  - As a replacement for std::atomic
  - As a qualifier for return types
  - As a qualifier on function parameters
  - To enforce ordering of non-volatile accesses

Designated Initializers


```
1. struct account {
2.    int number;
3.    int balance;
4.    int amount;
5. };
6.
7.
8. int main()
9. {
10.   account act;
11.   act.number = 16075;
12.   act.amount = 300;
13.}
```

```
1. int main()
2. {
3.    account act{.number = 16075, .amount = 300};
4. }
```

- Ordering matters (unlike in C)

using enum


```
1. enum class rgba_color_channel {
2.     red,
3.     green,
4.     blue,
5.     alpha
6. };
7.
8. std::string_view to_string(rgba_color_channel channel) {
9.     switch (channel) {
10.        case rgba_color_channel::red:    return "red";
11.        case rgba_color_channel::green:  return "green";
12.        case rgba_color_channel::blue:   return "blue";
13.        case rgba_color_channel::alpha:  return "alpha";
14.    }
15.}
```

using enum


```
1. enum class rgba_color_channel {
2.     red,
3.     green,
4.     blue,
5.     alpha
6. };
7.
8. std::string_view to_string(rgba_color_channel channel) {
9.     switch (channel) {
10.        using enum rgba_color_channel;
11.
12.        case red:    return "red";
13.        case green:  return "green";
14.        case blue:   return "blue";
15.        case alpha:  return "alpha";
16.    }
17.}
```

std::format


```
1. printf("String `%s' has %d characters\n", string, length(string)));
```

```
1. std::cout << "String `" << string << "' has " << length(string) << " characters\n";
```

```
1. std::string str = std::format("String `{}' has {} characters\n", string, length(string)));
```

Security Beyond C++20

- Contracts (pushed to C++23 at least)
- secure_clear (probably C++23)
- Zero-overhead Deterministic Exceptions (maybe C++23 but more likely C++26)
- Enumerating Core Undefined Behavior

- More focus on safety critical and security issues