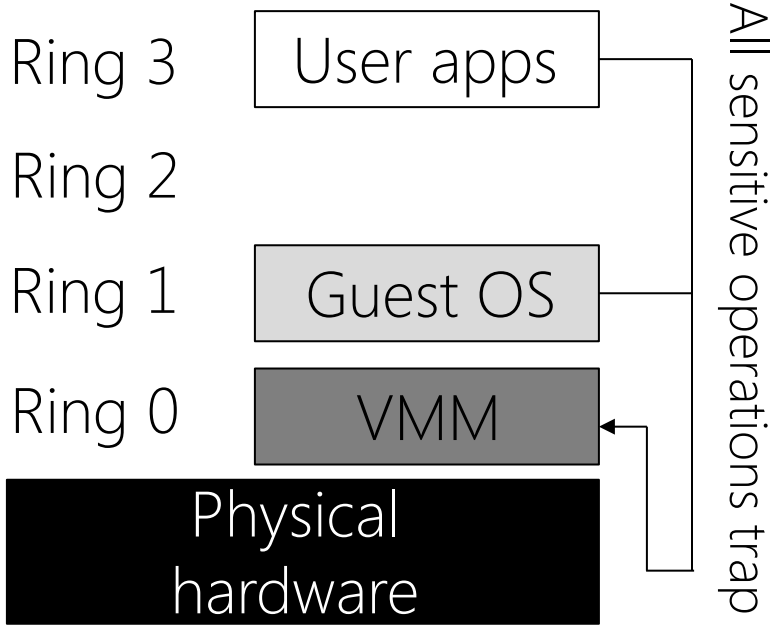
Security, Part II

Using Virtualization for Evil

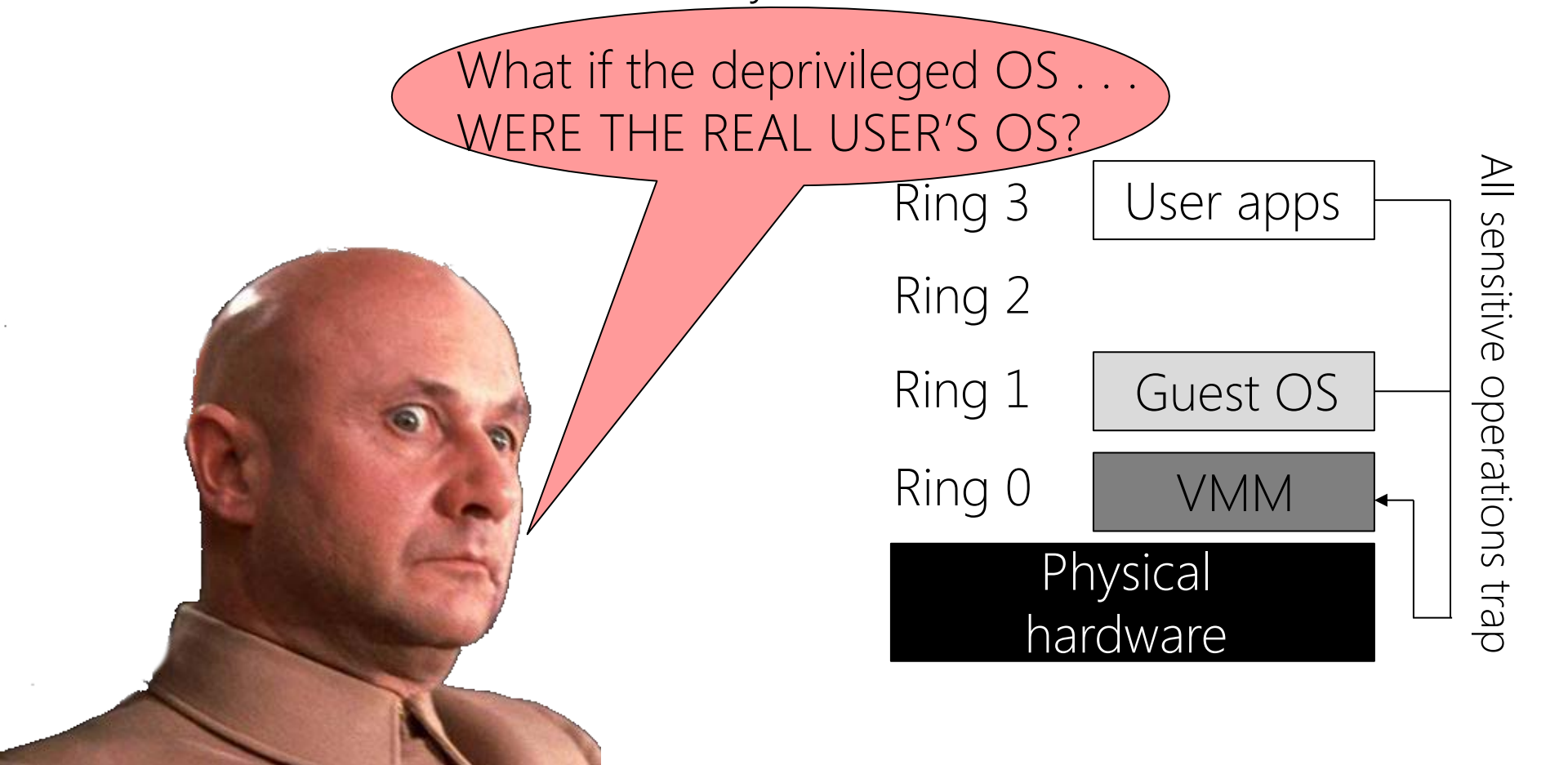
Deprivileging a Guest OS

- Using virtualization technology, we can mediate how a Guest OS (and its applications) interact with the outside world
  - Ex: Direct execution w/binary translation and a bare-metal VMM
  - VMM dynamically translates non-virtualizable instructions into virtualizable equivalents
  - As a result, all sensitive operations generate a trap into VMM; VMM can then decide whether to kill the Guest OS, perform the requested operation and return a (maybe modified) result, etc.



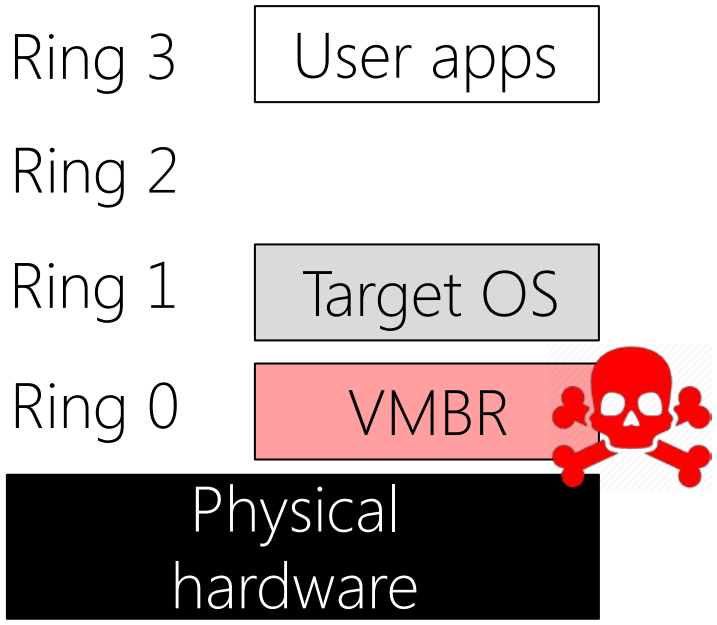
Deprivileging a Guest OS

- Using virtualization technology, we can mediate how a Guest OS (and its applications) interact with the outside world
  - Ex: Direct execution w/binary translation and a bare-metal VMM



Virtual-machine Based Rootkit (VMBR)

- Suppose that an attacker could hack into a target machine, and then move the user's OS+apps into a VM
  - VMM controls physical RAM, so the VMM can arbitrarily tamper with the victim's code and data
  - VMM controls physical devices, so the VMM can snoop on the victim's keystrokes, network traffic, disk IO
  - If VMM is well-implemented, the victim's code can't detect that it's running inside a VM!
- The fundamental principle: lower levels control higher levels!



When You Turn The Power Button On . . .

- The motherboard initializes itself and turns on a CPU to be the bootstrap CPU (the other CPUs are left idle at first)
- The CPU starts executing the Basic Input-Output System (BIOS)
  - BIOS is low-level software provided by the motherboard manufacturer
  - The motherboard maps the BIOS code (typically stored in Flash memory) to a well-known location in RAM, so that the CPU knows where to start execution (on x86, location is 0xF0000)
- BIOS probes for the existence of other hardware like hard disks, keyboards, etc., and then starts the boot sequence
  - BIOS reads the first sector of the hard disk which contains the Master Boot Record (MBR)
  - BIOS loads MBR into memory and jumps to that instruction
  - BIOS then loads the rest of the OS from disk

VMBR: Corrupting The Boot Sequence

- Step 1: Attacker gains the ability to execute code
  - Probably the easiest part of the attack
  - Phishing emails, Java vulnerability, drive-by download, etc.

UNINSTALL JAVA

IT'S KILLING YOU

DISABLE IT NOW

Attacking Java's Best-fit Mapping Conversions

- Best-fit mapping occurs when a program converts a string from encoding X to encoding Y, and Y does not have an exact representation for a particular character
- Java Web Start (JAWS) allows a user to launch a Java application by clicking on a link in a web browser
  - Whenever a program receives input from an untrusted source, that input should be sanitized (i.e., stripped of dangerous characters)
  - To prevent a website from passing malicious arguments to javaw.exe, JAWS quotes entire argument string, and then escapes any quotation marks inside of it

javaw.exe -arg="maliciousArg1 maliciousArg2 "maliciousArg3

Attacker-provided string in red is sanitized to . . .

javaw.exe "-arg=\" maliciousArg1 maliciousArg2 \"maliciousArg3"

Attacking Java's Best-fit Mapping Conversions

- Best-fit mapping occurs when a program converts a string from encoding X to encoding Y, and Y does not have an exact representation for a particular character
- Java Web Start (JAWS) allows a user to launch a Java application by clicking on a link in a web browser
  - Whenever a program receives input from an untrusted source, that input should be sanitized (i.e., stripped of dangerous characters)
  - To prevent a website from passing malicious arguments to javaw.exe, JAWS quotes entire argument string, and then escapes any quotation marks inside of it

javaw.exe -arg=[U+02BA] maliciousArg1 maliciousArg2 [U+02BA]maliciousArg3

Sanitizer only looks for 0x22, and then JAWS best-fits to ASCII . . .

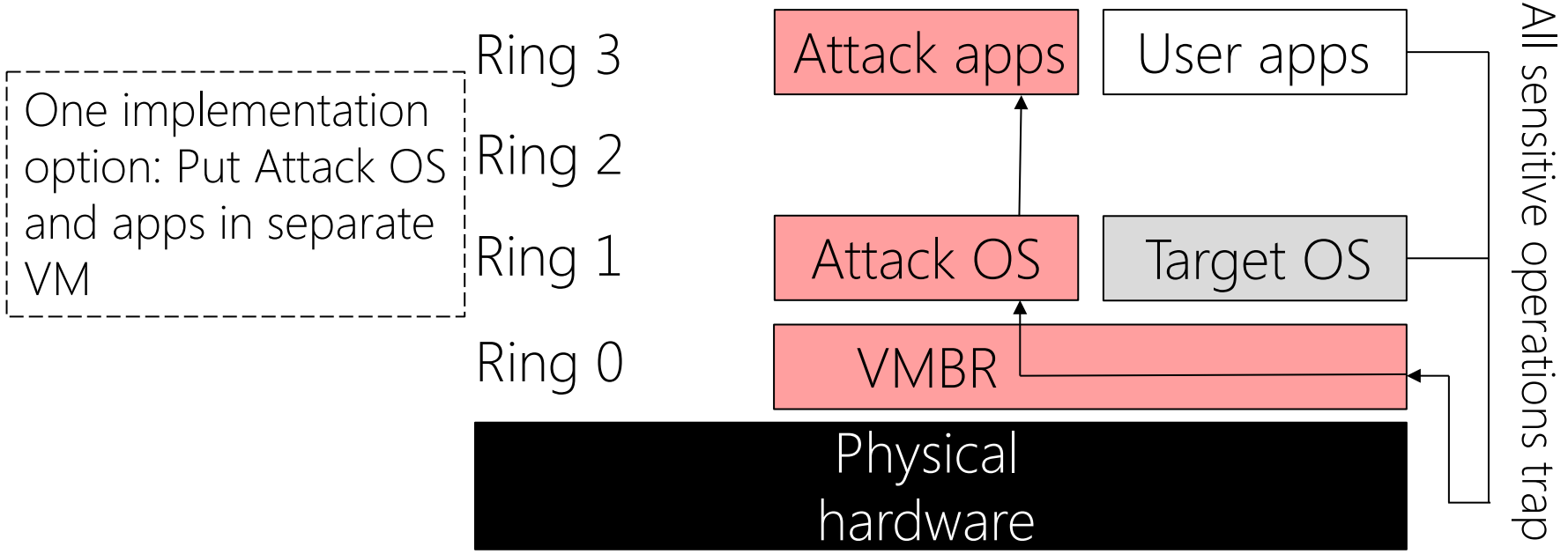
javaw.exe "-arg=" maliciousArg1 maliciousArg2 "maliciousArg3"

VMBR: Corrupting The Boot Sequence

- Step 1: Attacker gains the ability to execute code
  - Probably the easiest part of the attack
  - Phishing emails, Java vulnerability, drive-by download, etc.
- Step 2: Attacker downloads the VMBR
  - I was lying; this is actually the easiest part of the attack
- Step 3: Corrupt the disk state
  - In the first proof-of-concept VMBR on Linux, the attacker disables swapping and then stores the rootkit state in the swap space; leaves other on-disk blocks in their normal place
  - Definitely the trickiest part of the attack, since the attacker needs to do this without causing the kernel to crash, while simultaneously avoiding antivirus software (which looks for tampering of boot blocks!)
- Step 4: Reboot the machine
  - If the attacker has root privileges, then rebooting is easy

VMBR: After Post-infection reboot

- VMBR loads target OS in a virtual machine
  - Target OS's disk IO is redirected to a virtual disk that's controlled by the VMBR
- VMBR also creates an "attack OS" to manage the evil
  - Attack OS allows the malware to implement the attack control logic in user-mode code, and take advantage of device drivers and other OS functionality

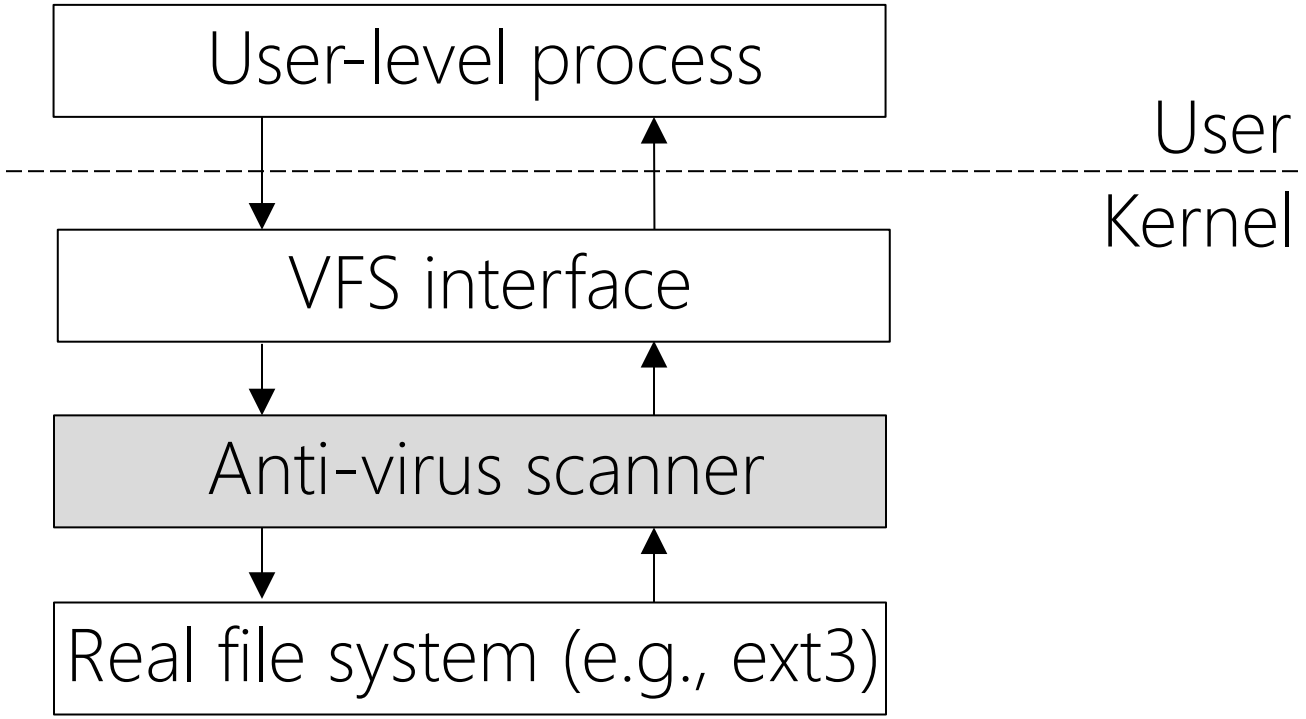


Can VMBR Be Detected?

- User can boot from a safe medium like a CD-ROM or USB drive, use a recovery program to detect the malicious boot data on disk
  - But why would the user think to do this? Perf problems?
- Virtualization adds overhead via trap-and-emulate
  - Target OS can measure the speed of benchmarks that use sensitive instructions
  - However, if VMBR uses hardware-assisted virtualization (e.g., Intel VT-x), the attacker can virtualize timestamp instructions (e.g., RDTSC on x86)!
- Target OS can try to look for "virtualization artifacts"
  - Ex: VMWare-specific paravirtualization network drivers placed in the target OS
  - However, an attacker which uses a customized VMM will avoid leaving such artifacts

What About Anti-virus Scanners and Network Intrusion Detectors?

- Anti-virus scanners are typically implemented as a file system driver (Windows) or "stackable file system" (Linux)



- Network intrusion detectors are similarly implemented as filter drivers
- Both types of defensive mechanisms live in the kernel . . .
  - . . . and thus cannot detect evil at the VMBR level!

The “Blue Pill” Kerfuffle

- Blue Pill was a proof-of-concept VMBR that used AMD/x86 hardware-assisted virtualization
- "There is no software-visible bit whose setting indicates whether a logical processor is in VMX non-root operation. This fact may allow a VMM to prevent guest software from determining that it is running in a virtual machine." –Intel VT-x specification
- Inventor of Blue Pill claimed that since Blue Pill uses hardware-assisted virtualization (including RDTSC), Blue Pill is "100% undetectable"

The "Blue Pill" Kerfuffle

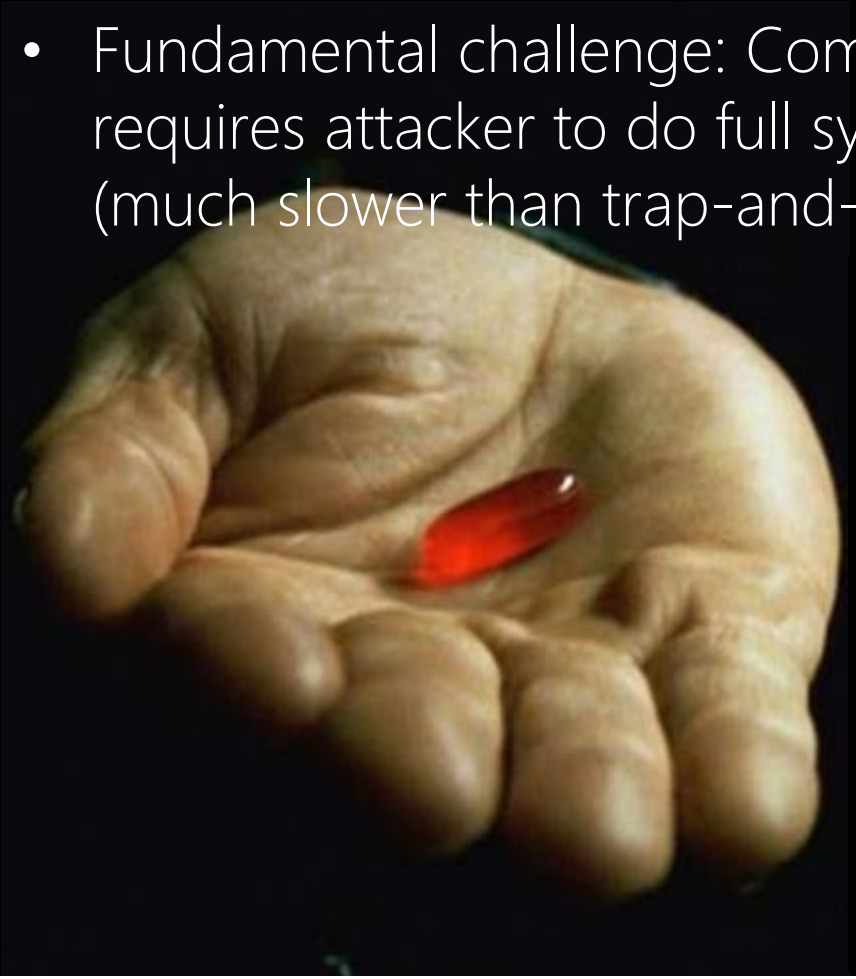
- Security researchers started looking for “red pills,” i.e., mechanisms that target OSes can use to detect Blue Pill

  - One commonly proposed red pill: external sources of time
    - VMBR can virtualize local clock interfaces, but can’t virtualize clocks on remote machines!
    - So, target OS can try to time sensitive instructions w.r.t. time stamps fetched over the network
- Timing complication 1: VMBR controls network interfaces, so if it knew the network timing protocol, could it rewrite packets? [Seems hard]
- Timing complication 2: How much jitter is involved in received network packets? Will this confuse timing measurements? [VMBR can inject jitter—very easy]

- Many people also claimed that properly virtualizing RDTSC is impossible for the VMBR to do correctly

  - An engineer at VMware called the Blue Pill research "quasi-illiterate gibberish"
  - "The claim is that SVM/VT make it possible to cloak the presence of a VMM rootkit completely. Allow me to go on record: this claim is pure fantasy. In practice, it is always possible to detect the presence of a VMM, via timing attacks."
- "VMware's software goes to extremely clever lengths in coordinating and manipulating the various virtual time sources just to get certain versions of the Linux kernel to boot . . . This was the result of a real, naturally occurring guest code that had nothing to do with VM detection; you can imagine that a dedicated attack on the VMM's virtual time sources could be many orders of magnitude more effective."

- Fundamental challenge: Completely accurate timing virtualization requires attacker to do full system simulation via interpretation (much slower than trap-and-emulate!)



- Ex: Does the VMBR emulate the relative latencies of a TLB miss versus polling the network card for data? If the VMBR doesn't do both, target OS can detect the timing discrepancy!
- Cycle-accurate simulation has a performance overhead of ~25,000x; users would notice!

Home / Security

The Undetectable Malware That Real Hackers Don't Seem to Want

COMMENTS

By Robert Mcmillan, IDG News Service
Aug 5, 2011 2:16 PM

- Given the complexity of VMBRs, they don't seem worth the trouble for most attackers

Attested Booting: iPhone
. . . but first, public key cryptography

- Each participant Alice in the cryptosystem has a public key and a private key
  - Alice distributes $PubKey_{Alice}$ to people who want to verify that Alice has sent them messages
  - Alice keeps $PrivKey_{Alice}$ private
- The signature function sig(msg, key) has a special property . . .

$$sig(sig(msg, PrivKey_{Alice}), PubKey_{Alice}) = msg$$

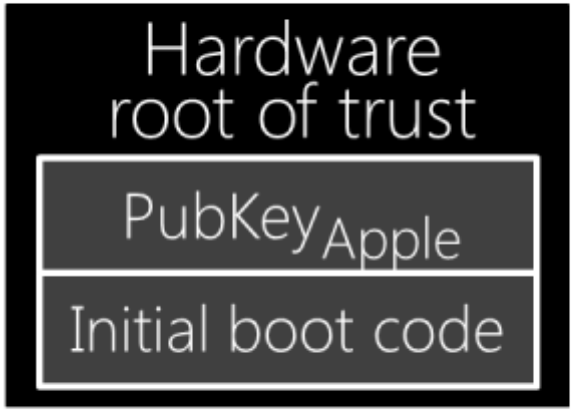
. . . but only if the public/private keys match

- Alice can send <msg, sig(msg, $PrivKey_{Alice}$)> to Bob, allowing Bob to use $PubKey_{Alice}$ to verify the signature
  - Optimization: Alice sends <msg, sig(hash(msg), $PrivKey_{Alice}$)>
  - hash(msg) takes an arbitrary length msg and outputs a fixed-length output (say, 20 bytes)
  - A good hash function should be uniform (i.e., spreading all inputs evenly and randomly across output space)

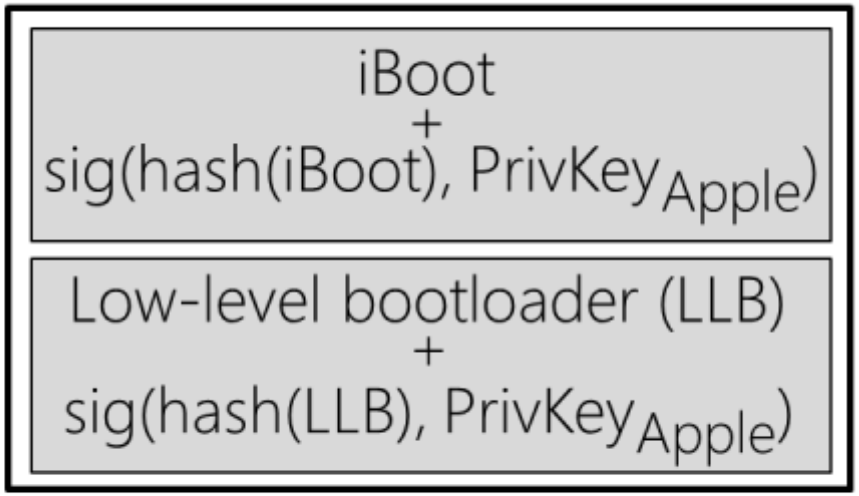
Attested Booting: iPhone

NOR storage is more expensive, but faster to read and supports byte-level addressing.

Updatable NAND Flash



Immutable ROM

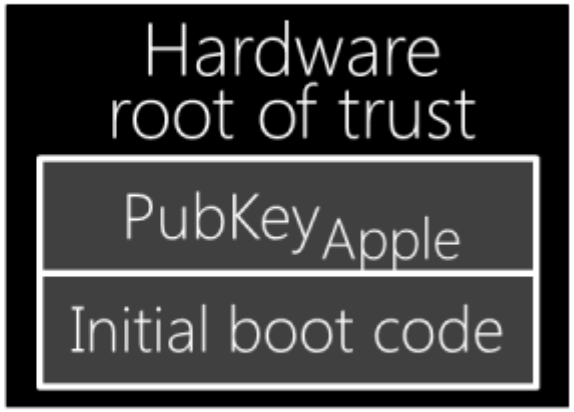


Updatable NOR Flash

Attested Booting: iPhone

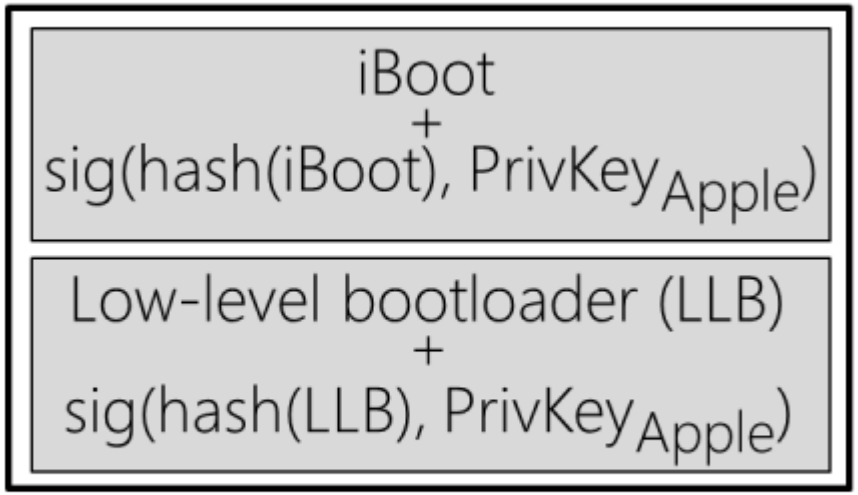
- Initial boot code loads LLB, validates LLB's signature, jumps to LLB if validation succeeds
- LLB loads iBoot, validates iBoot's signature, jumps to iBoot if validation succeeds
- Attestation continues recursively, allowing iPhone to validate entire software stack running on hardware!

Third party needs a public key signed by Apple, so that phone knows that Apple likes the third party.



Immutable ROM

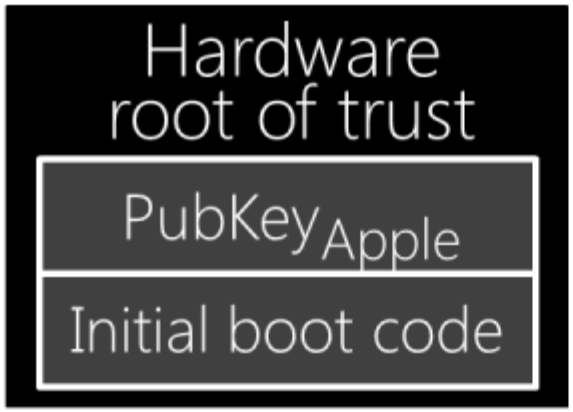
Updatable NAND Flash



Updatable NOR Flash

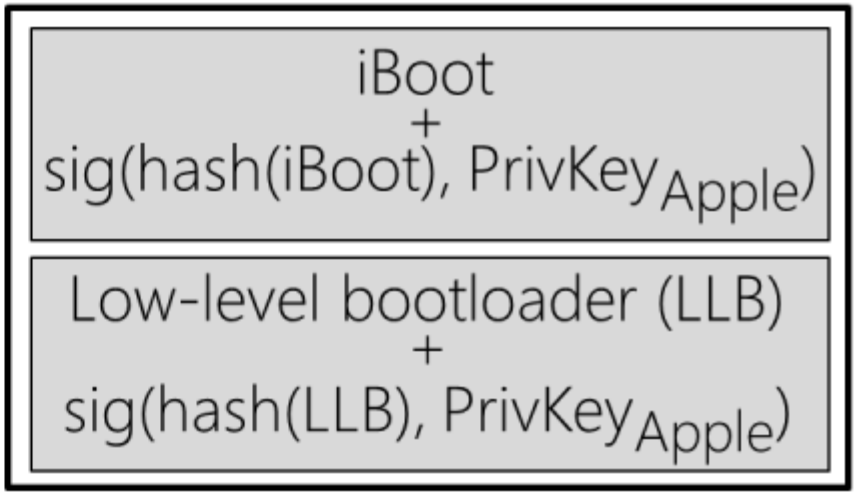
Attested Booting: iPhone

- Attested boot allows Apple to control what software users run
  - "Jailbreak" means "convince the iPhone to load software that wasn't signed by Apple or an Apple-blessed 3rd party key."
  - VMBR won't have a valid signature, so VMBR requires a jailbreak to work—an arbitrary privilege escalation is insufficient



Immutable ROM

Updatable NAND Flash



Updatable NOR Flash

Formally Verified Software

- If you could verify the correctness of your code, you could attest to a formally correct software stack!

```
//Example written in the Dafny language
//(a language which is designed to be
//easily verified)
method Abs(x: int) returns (y: int)
    ensures y >= 0;
    ensures x >= 0 ==> y == x;
    ensures x < 0 ==> y == -x;
{
    //Here's the code to write.
}
```

Formally Verified Software

- If you could verify the correctness of your code, you could attest to a formally correct software stack!

```
//Example written in the Dafny language
//(a language which is designed to be
//easily verified)
method Abs(x: int) returns (y: int)
    ensures y >= 0;
    ensures x >= 0 ==> y == x;
    ensures x < 0 ==> y == -x;
{
    y := -x;
}
```

Compilation fails!

Formally Verified Software

- If you could verify the correctness of your code, you could attest to a formally correct software stack!

```
//Example written in the Dafny language
//(a language which is designed to be
//easily verified)
method Abs(x: int) returns (y: int)
    ensures y >= 0;
    ensures x >= 0 ==> y == x;
    ensures x < 0 ==> y == -x;
{
    if(x < 0){
        y := -x;
    }else{
        y := x;
    }
}
```

Compilation succeeds!

Formally Verified Software

- Basic idea: Compile your program into a formula, prove that the formula satisfies certain properties, and if so, compile the formula to assembly code
  - Analyze formula using SMT solver ("satisfiability modulo theory")
  - SMTs are a type of logical formula, and the SMT solver tries to solve a constraint satisfaction problem
- Open questions:
  - What's the best way to reason about time and asynchrony?
  - Is the programmer effort spent writing (formal, but maybe still buggy) specs more or less than effort for writing (certainly incomplete, but maybe good enough) tests?
  - Should we design hardware to be more amenable to verification? [Formally verifying software requires a formal model for the underlying hardware—x86 OH NO, but maybe you just need to formally model an x86 subset?]
- See the Ironclad paper for more details [Hawblitzel 2014]