The Young Pioneers

Robert Grimm
New York University
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
The Challenge

* Ensure integrity of binaries
  * Make sure the code loaded is the right code
* Ensure integrity of execution
  * Make sure the code running is the right code
Trusted Computing

Basic idea
- Establish root of trust: load code, verify execution
  - Apply recursively across OS, applications
- Attest to chain of trust
  - To outside monitor
    - Think organizational intrusion detector
  - To running application
    - Think digital rights management (DRM)

Approaches
- Hardware: tamper-resistant chip with secret key (TPM)
- Software: challenge-response protocol (Pioneer)
2.3 Attacker Model

We assume that the adversary does not load malicious firmware onto the untrusted platform including the OS. However, we assume that the adversary has administrative privileges and can tamper with all software on the untrusted platform. For example, the adversary does not load malicious firmware onto the untrusted platform. The adversary can independently verify the checksum. Also, since the dispatcher has a copy of the verification function and patcher. The dispatcher has a copy of the verification function and code invokes the send function to return the checksum to the dispatcher. The untrusted platform uses the checksum code to ensure that the send function, the hash function and the executable can execute untampered. After computing the checksum, the checksum ensures that the send function, the hash function and the executable can execute untampered. The checksum code also sets up an execution environment to enforce an integrity measurement on the executable, setting up an execution environment for the executable. As Figure 1 shows, the verification function has three parts: a checksum code, a hash function and a send function.

The protocol has two steps. First, the dispatcher obtains an assurance that there is a dynamic root of trust on the untrusted platform. In other words, the dispatcher must be sure that the send function, the hash function and the executable can run untampered by any malicious software. After obtaining the assurance, the dispatcher first sends a challenge containing a random nonce to the untrusted platform. The untrusted platform is responsible for performing an integrity measurement of the executable. Although the collision resistance property of SHA-1 has been compromised, we rely on the second-preimage collision resistance property for which SHA-1 is considered strong. We take advantage of the compromised collision resistance property of SHA-1 to create two different copies of the executable both of which have the same hash value. After the measurement, the hash function takes advantage of the compromised collision resistance property of SHA-1 to create two different copies of the executable both of which have the same hash value. The adversary cannot create two different copies of the executable both of which have the same hash value. The adversary cannot forge the correct checksum value in spite of having modified the verification function. For example, the adversary tries to manipulate the checksum computation, the checksum computation to forge the correct checksum value in spite of having modified the verification function. The checksum code also sets up an execution environment to enforce an integrity measurement of the executable. The checksum code also sets up an execution environment to enforce an integrity measurement of the executable.

3.2 The Pioneer Protocol

We use SHA-1 as the hash function to perform the integrity measurement of the executable. Although the collision resistance property of SHA-1 has been compromised, we rely on the second-preimage collision resistance property for which SHA-1 is considered strong. We take advantage of the compromised collision resistance property of SHA-1 to create two different copies of the executable both of which have the same hash value. After the measurement, the hash function takes advantage of the compromised collision resistance property of SHA-1 to create two different copies of the executable both of which have the same hash value. The adversary cannot create two different copies of the executable both of which have the same hash value. The adversary cannot forge the correct checksum value in spite of having modified the verification function. The adversary tries to manipulate the checksum computation, the checksum computation to forge the correct checksum value in spite of having modified the verification function. For example, the adversary tries to manipulate the checksum computation, the checksum computation to forge the correct checksum value in spite of having modified the verification function.
Assumptions and Attackers

Assumptions
- Dispatchers knows exact hardware configuration
- Network provides message-origin authentication
- Untrusted platform only communicates w/dispatcher

Attacker
- Has complete control over software
- Does not modify hardware nor firmware
- Does not issue DMA requests
Checksum Code

- Time-optimal implementation
  - Use instructions that are hard to replace, implement faster
    - add, xor
  - Ensure that operations in one block depend on previous
- Instruction sequencing
  - Eliminate empty issue slots on superscalar CPUs
Checksum Code (2)

- CPU state inputs
- Avoid memory copy attack
- Include PC and data pointer in checksum
- Known (virtual) address

![Diagram of checksum code with four attack scenarios:](image-url)
Iterative checksum code

- Each iteration incorporates one memory location
- Adversary incurs constant time overhead per iteration

### Verification Function

<table>
<thead>
<tr>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum Initialization Code</td>
</tr>
<tr>
<td>Checksum Loop</td>
</tr>
<tr>
<td>Epilog Code</td>
</tr>
<tr>
<td>Send Function</td>
</tr>
<tr>
<td>Hash Function</td>
</tr>
</tbody>
</table>
Checksum Code (4)

- **Strongly-ordered function**
  - Output differs if instructions are executed in different order
  - Example: $a_1 \oplus a_2 + a_3 \oplus a_4 + a_5$
  - Prevents parallelization
  - Prevents re-ordering

- **Small code size**
  - Ensures fast execution time since code fits into cache
  - Ensures adversary’s overhead is more noticeable
Low variance of execution time

- Helps set threshold value through 3 techniques
  - Execute at highest CPU privilege level
    - No other code can run
  - Fit code into L1 instruction and data caches
  - Sequence instructions to fill superscalar issue slots
    - Well, not quite in real life
Checksum Code (6)

- Keyed-checksum
  - Prevents precomputing of checksum
    - Seed pseudo-random number generator used in loop
      - T-function: $x \leftarrow x + (x^2 \lor 5) \mod 2^n$
    - Initialize checksum value
- Pseudo-random memory traversal
  - Prevents data substitution attack
  - Read $O(n \ln n)$ locations to cover entire verification function with high probability (Coupon Collector’s Problem)
Are We There Yet?
No: Execution Environment

- Highest privilege level with interrupts turned off
  - Prevent other code, including VMM from running
  - Incorporate status register into checksum
    - Includes carry & zero flags, interrupt flag
- What about non-maskable interrupts?
  - Disable in epilog (why?)
  - Detect previous events by placing checksum on stack
    - Would be clobbered by interrupt handler

![Diagram of stack pointer range](image-url)

Low Address | High Address
------------|--------------
Word 0      | Word 5       
Word 1      |               
Word 2      |               
Word 3      |               
Word 4      |               
Word 5      |               

Stack Pointer Range

Stack Pointer
Are We There Yet?
With the Design, Yes!
x86 Implementation

- Indirectly include CPU state through absolute jumps
  - PC not available as operand
- Avoid memory copy attacks
  - By using all registers to prevent register-based offsets
  - By randomly jumping to prevent immediate offsets
    - Adversary would have to space code blocks
- Space out checksum on stack to use rflags

![Stack Layout Diagram](image)
We observe the following points: 1) even the running time of the trusted platform for the adversary's checksum computation is the deterministic runtime; 3) we observe some false positives (5 out of)

The dispatcher sends another challenge. The untrusted platform re-

Our LAN segment. We run our experiments for 2 hours at each lo-

Theor

1,250,000. To prevent false pos-

We suggest two methods for RTT estimation. First, the dis-

Figures 12 and 13 show the results of our experiments at the

With all exceptions are disabled by the processor, 2) paging and virtual

Virtual-memory-based attacks.

It is easier to defend against the segmentation-based memory copy

System Management Mode (SMM) is a special operating mode

We now discuss virtual-memory-based attacks, issues concerning

Execution Environment

Code running in the SMM mode runs

Theoretically Best Adversary’s Runtime

Adversary’s Runtime and Network RTT

Location 1

Location 2

False Positives
What About False Positives?

- Pioneer runtime has very little variation
- But RTT has larger variation
- Therefore, need to measure RTT more frequently and for individual machines (not entire network)
  - Just before sending challenge
  - Continuously for all machines
Rootkit detector for Fedora Core 2 version of Linux

- Verifies kernel, incl. text segment, syscall table, modules

1. Introduction: “Our rootkit detector was able to detect all publically-known rootkits for this series of the Linux kernel”

6. Applications: “successfully detected adore-ng-0.53, the only publically-known rootkit for the 2.6 version of the Linux kernel”

Editorial: What a difference 14 pages make...
What Do You Think?