TrustLogin: Securing Password-Login on Commodity Operating Systems

Fengwei Zhang
George Mason University
Fairfax, VA, USA
fzhang4@gmu.edu

Kevin Leach
University of Virginia
Charlottesville, VA, USA
kjl2y@virginia.edu

Haining Wang
University of Delaware
Newark, DE, USA
hnw@udel.edu

Angelos Stavrou
George Mason University
Fairfax, VA, USA
astavrou@gmu.edu

ABSTRACT

With the increasing prevalence of Web 2.0 and cloud computing, password-based logins play an increasingly important role on user-end systems. We use passwords to authenticate ourselves to countless applications and services. However, login credentials can be easily stolen by attackers. In this paper, we present a framework, TrustLogin, to secure password-based logins on commodity operating systems. TrustLogin leverages System Management Mode to protect the login credentials from malware even when OS is compromised. TrustLogin does not modify any system software in either client or server and is transparent to users, applications, and servers. We conduct two study cases of the framework on legacy and secure applications, and the experimental results demonstrate that TrustLogin is able to protect login credentials from real-world keyloggers on Windows and Linux platforms. TrustLogin is robust against spoofing attacks. Moreover, the experimental results also show TrustLogin introduces a low overhead with the tested applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; B.4.2 [**Hardware**]: Input/Output Devices—*Channels and controllers*

Keywords

Login Password; System Management Mode; Keyloggers

1. INTRODUCTION

Logging in is part of daily practice in the modern world. We use it to authenticate ourselves to applications for resource accesses. Consequently, login credentials are one of the top targets for attackers. For example, keylogger malware found on UC Irvine health center computers in May 2014, and it is estimated that 1,813 students and 23 non-students were impacted [7]. Additionally, it is reported that attackers have stolen credit card information for customers who shopped at 63 Barnes & Noble stores using keyloggers [3]. A case study has shown that 10,775 unique bank account credentials were stolen by keyloggers in a seven-month period [22]. Protecting login credentials is a critical part of daily life.

Nowadays, operating systems are complex and rely on millions of lines of code to operate (e.g., the Linux kernel has about 17 million lines of code [35]). The large Trusted Computing Base (TCB) of these OSes inevitably creates vulnerabilities that could be exploited by attackers. The Common Vulnerabilities and Exposures (CVE) list shows that 240 vulnerabilities have been found for the Linux kernel [2]. An attacker can easily leverage these vulnerabilities to create rootkits and keyloggers.

On top of an untrusted OS, no matter how secure the network applications are, the sensitive data used by secure applications is at risk of leakage. For example, an attacker can install a stealthy keylogger after compromising the OS, so the banking login information entered in a web browser can be obtained by the attacker without a user awareness. Therefore, the protection of the user's sensitive data during network operations is crucial; we need to prevent malicious behaviors of attackers on network applications.

In this paper, we present TrustLogin, a framework to securely perform login operations on commodity operating systems. Even if the operating system and applications are compromised, an attacker is not able to reveal the login password from the host. TrustLogin leverages System Management Mode (SMM), a CPU mode that exists in x86 architecture, to transparently protect the login credentials from keyloggers. Since we assume the attackers have ring 0 privilege, all of the software including the operating system cannot be trusted. SMM is a separate CPU mode with isolated execution memory, and it is inaccessible from the OS, which satisfies the needs of our system.

When users enter their passwords, TrustLogin automatically switches into SMM and records the keystroke. It provides a randomly generated string to the OS kernel and then the network driver prepares the login packets. When the login packets arrive at the network card, TrustLogin switches

ASIA CCS'15, April 14–17, 2015, Singapore..
Copyright © 2015 ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714614 .

into SMM again and replaces the placeholder with the real password. Under the protection of TrustLogin , rootkits (e.g, keyloggers) cannot steal the sensitive data even with ring 0 privilege. To defend spoofing attacks, we implement two novel techniques that ensure the trust path when switching to SMM. They use the LED lights on keyboard and the PC speaker to interact with users. More importantly, TrustLogin does not modify application- and OS-code, and it is transparent from client and server sides.

To demonstrate the effectiveness of our approach, we conduct two study cases to use TrustLogin with legacy and secure applications. We test TrustLogin with real-world keyloggers on both Windows and Linux platforms, and the experiment results show that TrustLogin is able to protect the login password against them. We also measure the performance overhead introduced by executing code in SMM. Our results show SMM switching only takes about 8 microseconds. TrustLogin takes 33 milliseconds to store and replace a keystroke and most of the time is consumed by trusted path indication (i.e., playing a melody and showing a LED light sequence); it spends 30 microseconds on injecting the password back to a login packet for the tested application. The contributions of this paper are summarized as follows:

- We propose a framework, TrustLogin, to secure passwords when logging in on commodity operating systems. It leverages System Management Mode to protect the sensitive data from exposing to rootkits and thus guarantees the security on the local host.
- TrustLogin does not modify application- and OS- code, and it is transparent from both user-ends and servers. TrustLogin does not rely on hypervisor or OS code, and it introduces a minimal trusted code.
- We implement our framework on legacy and secure applications. We also demonstrate that TrustLogin can prevent real-world keyloggers from stealing passwords on Windows and Linux platforms.
- TrustLogin is robust against spoofing attacks by ensuring the trusted path of SMM switching. The performance experiments show that TrustLogin is lightweight and efficient to use.

The rest of the paper is organized as follows. Section 2 explains the background of the work. We discuss the threat model and assumptions in Section 3. Section 4 outlines the system architecture and implementation of TrustLogin. We propose the login case studies in Section 5. The discussion and limitation of TrustLogin is explained in Section 6. We present the related work in Section 7. Section 8 concludes the paper and discusses future work.

2. BACKGROUND

2.1 Advanced Programmable Interrupt Controller

The Advanced Programmable Interrupt Controller (APIC) is used to handle the communication between CPU and peripherals. There are two components in the Intel APIC system, the Local APIC (LAPIC) and the I/O APIC. The LAPIC performs two primary functions for the processor: 1) It receives interrupts from an external I/O APIC and sends these to the processor core for handling; 2) it sends and receives Interprocessor Interrupt (IPI) messages to and from other logical processors. The LAPIC is integrated in the CPU and each processor core has a LAPIC. The external I/O APIC is part of the system chipset. Its primary function is to receive the interrupts from I/O devices and forwards them to the LAPIC as interrupt messages. Normally each peripheral bus has an I/O APIC. In TrustLogin, we reconfigure APIC to generate SMIs.

2.2 System Management Mode

System Management Mode (SMM) is a CPU mode available in all x86 architecture. It is similar to Real and Protected Modes. Originally, it was designed for implementing system control functions such as power management. In recent years, it has been used for system introspection, debugging and so on. SMM is implemented by the Basic Input/Output System (BIOS). Before the system boots up, the BIOS loads SMM code into System Management RAM (SMRAM), a special memory region that is inaccessible from any other CPU mode. SMM is triggered by asserting the System Management Interrupt (SMI) pin on the motherboard. Both hardware and software are able to assert this pin, although the specific method depends on the chipset. After assertion, the system automatically saves its CPU states into SMRAM, and then executes the SMI handler code. A `RSM` instruction is executed at the end of the SMI hander to switch back to Protected Mode. In TrustLogin, we use SMM as a trusted execution environment to implement critical operations.

3. THREAT MODEL AND ASSUMPTIONS

3.1 Keylogger

Keyloggers can be classified into two types: hardware- and software-based. Hardware-based keyloggers are small electronic devices that are used to capture the keystrokes. They are often built in the keyboard itself and have separate non-volatile memory to store the keystrokes. Hardware keyloggers do not require installation of any software or power source for their operations. For instance, there are some commercial hardware keyloggers available [8]. In this paper, we do not consider this type of keylogger, and we assume the keyboard is not malicious.

Software-based keyloggers are installed within the operating system, and most of the keyloggers in the real world are this type. There are two kinds of software keyloggers: user- and kernel-level. For instance, a user-level keylogger can use the `GetKeyboardState` API function to capture keystrokes in Windows. This kind of keylogger is efficient but also easily detected. Kernel-level keyloggers are implemented at the kernel level and require administrator privilege to install. For example, a keyboard filter driver can be used to stealthily capture keystrokes [37]. TrustLogin considers software-based keyloggers as the threat model, and it guarantees that keystrokes cannot be stolen if such a keylogger is present.

3.2 Assumptions

TrustLogin assumes that the attackers have unlimited computing resources and can exploit zero-day vulnerabilities of the host OS and desktop applications. We only consider attacks against the host machine; network attacks are out of the scope of this paper. We do not consider phishing at-

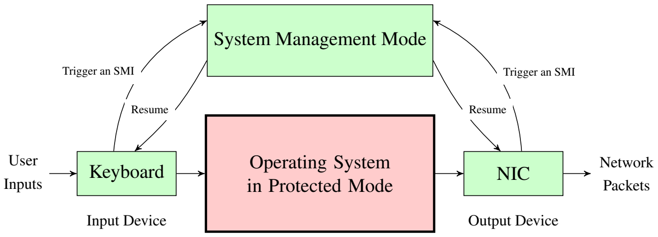


Figure 1: Architecture of TrustLogin

tacks, which trick users to send their credentials to a remote host. We assume the hardware and firmware of the host machine are trusted, and the attacker cannot flash the BIOS or modify the firmware. We assume SMRAM is locked and remains intact after boot, and the attacker cannot change the SMI handler. We assume that the attacker does not have physical access to the machine. We do not consider Denial-of-Service (DoS) attacks against our system. Ring 0 malware can easily disable SMI triggering and stop the login process.

4. SYSTEM FRAMEWORK

Figure 1 shows the architecture of TrustLogin. There are four rectangles in the figure; the green rectangles represent trusted components, including the keyboard, Network Interface Card (NIC), and System Management Mode (SMM). The red rectangle represents the operating system in Protected Mode, which may have been compromised by attackers. When a user inputs the sensitive information (e.g., password) from the keyboard, the keyboard automatically triggers an SMI for every key press. The SMI handler (which executes in SMM) records the keystrokes and inserts bogus place-holders in the keyboard buffer. After resuming Protected Mode, the OS only handles the place-holders. In other words, the attackers with ring 0 privilege can only retrieve the string of place-holders. When the login packet is about to transmit, TrustLogin triggers another SMI by using the network card. The SMI handler replaces the bogus place-holders with the original keystrokes in the network packet. We also make sure the packet leaves the network card within the SMI handler so that malware cannot read the packet. Next, we explain the system step by step.

4.1 Entering Secure Input Mode

In TrustLogin, we have two modes for the system. One is the secure input mode, and the other one is the normal input mode. In the secure input mode, TrustLogin intercepts all of the keystrokes and protect them from the keyloggers or rootkits. When the user is about to enter the sensitive information (e.g., password), he or she needs to switch to the secure input mode. Past systems have used a variety of ways to notify the system. For instance, Bumpy [34] uses "@@" as a Secure Attention Sequence (SAS) to signal the system that the user is about to enter sensitive inputs.

One requirement of switching into the Secure Input Mode is that the entering method should be rarely used by default. Ideally, it should be unique (e.g., a dedicated hardware switch [40]), but SAS-like "@@" sequence also works. The other requirement is usability. In TrustLogin, we simply use the key combination, `Ctr+Alt+1`, to signal our system and enter the secure input mode. When TrustLogin reads an Enter key in the secure input mode, it stops intercepting keystrokes and switches to the normal input mode. Since users often end password inputs by pressing Enter, this is reasonable.

4.2 Intercepting Keystrokes

TrustLogin intercepts every keystroke and records them in the SMRAM in the secure input mode. Before introducing how keystrokes are intercepted in TrustLogin, we will explain how keystrokes are handled normally.

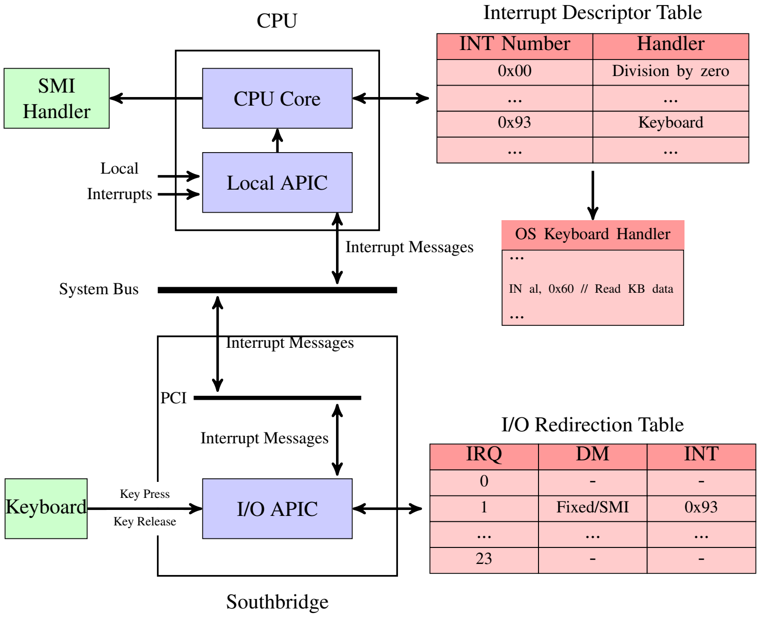
The input/output devices (e.g., keyboard) connect to the Southbridge (a.k.a. I/O controller Hub). Whenever a key is pressed or released, the keyboard notifies the I/O Advanced Programmable Interrupt Controller (APIC) in the Southbridge. I/O APIC looks up the I/O redirection table based on the Interrupt Request (IRQ), and then creates the corresponding interrupt message. The IRQ for the keyboard is 1, and the interrupt message includes the Delivery Mode (DM), Fixed, and the interrupt vector, `0x93`. The interrupt message goes through the PCI and system buses, and arrives at the local APIC in the CPU. Based on the DM and interrupt vector, the local APIC looks up the Interrupt Descriptor Table (IDT), and then the CPU jumps to the base address of the OS keyboard interrupt handler. The OS keyboard interrupt handler starts to execute keyboard handling functions. Specifically, it reads the keyboard data registers by accessing port `0x60` and may display the key value on the display monitor. Figure 2 shows the keystroke handling process.

Note that there are two interrupts for each keystroke: key press and key release. When the key is pressed or released, the keyboard sends a message known as "scan code" to the keyboard controller output buffer that the OS handler read later. There are two different types of scan codes: "make codes" and "break codes." A make code is sent when a key is pressed, and a break code is sent when a key is released. Every key has a unique make code and break code. There are three different sets of scan codes. Our keyboard uses the scan code set 1 [6]. For example, the make code of the `A` key is `0x1E`, and its break code is `0x9E`. If a user keeps holding the key, the keyboard would continue to send interrupts with the make code. When the user releases the key, the break code would be written into the output buffer of the keyboard controller.

To record the keystrokes, TrustLogin raises an SMI during the key press or release handling process and saves the keystrokes in the SMRAM. Additionally, we also save the scan code set mapping in the SMI handler to figure out which key is pressed or released. Next, we explain two approaches that we implement to trigger an SMI during the keystroke handling.

4.2.1 I/O Trap Approach

TrustLogin can use hardware I/O traps to generate an SMI [12]. The I/O trap feature allows SMI trapping on access to any I/O port using `IN` or `OUT` instruction. As mentioned, the OS interrupt handler needs to read the keyboard data register by accessing port `0x60`. If we configure the SMI I/O trap, an SMI would be triggered when the OS handler reads the keyboard data registers. In this way, we are able to intercept all keystrokes and save them in the SMRAM. When the OS keyboard interrupt handler



Interrupt Descriptor Table

INT Number	Handler
0x00	Division by zero
...	...
0x93	Keyboard
...	...

```
...
IN al, 0x60 // Read KB data
...
```

I/O Redirection Table

IRQ	DM	INT
0	-	-
1	Fixed/SMI	0x93
...	...	...
23	-	-

Figure 2: Keystroke Handling in TrustLogin

executes `IN al, 0x60` instruction, the system automatically generates an SMI. However, this `IN` instruction would not be executed again when resuming the OS in Protected Mode. To address this problem, the SMI handler needs to read the key value from the keyboard data register and store it in the `EAX` as if no trap has been created. Additionally, we need to disable the SMI I/O trap in the SMI handler. Otherwise, an SMI will be buffered when accessing the I/O port. In that case, the SMI will be immediately triggered after exiting SMM and the system will halt. Note that Wecherowski demonstrated similar triggering approach in [43].

4.2.2 I/O APIC Rerouting Approach

The other approach of triggering SMIs is to reroute keyboard interrupt by reconfiguring I/O APIC. As shown in Figure 2, The Delivery Mode (DM) of the I/O redirection table can be configured as "SMI" instead of "Fixed" normally. In other words, we are able to deliver an SMI to the CPU for every keyboard interrupt; that is, every key press causes our code to execute in SMM. Next, we store the keystroke to the SMRAM by reading the keyboard data register in the SMI handler.

We read the 1-byte scan code from the keyboard data register by reading I/O port `0x60`. After extracting the scan code, we map the scan code to the key value using the scan code set 1 table. Next, we store the key in the SMRAM. Since this approach reroutes the keyboard interrupt to an SMI, the original keyboard interrupt is not handled. To address this problem, we configure the keyboard control register (i.e., IO port `0x64`) to reissue the interrupt. We write the command code, 0xD2, to the control register. This special command means the next byte written to the keyboard data register (i.e., I/O port `0x60`) will be as if it came from the keyboard [42]. We write a replaced scan code back to the data register after writing the command code. After exiting SMM, another interrupt is generated due to the new data in the keyboard data register. Additionally, we need to disable SMI triggering in I/O APIC when reissuing the interrupt in the SMI handler. This makes sure that the reissued interrupt is a normal keyboard interrupt with "fixed" as the DM. Otherwise, an immediate SMI will be generated after exiting SMM, which causes an infinite loop (deadlock). The method for accessing the I/O APIC or keyboard controller is specified in the Southbridge datasheet [42].

Embleton et al. also used a similar approach to generate SMIs in [20]. However, we did not see that the I/O read operation of the keyboard data register was destructive in the SMI handler; we were able to read the data register multiple times until a new value was written. Additionally, it uses interprocessor interrupts (IPI) to reissue the interrupt by configuring the Interrupt Command Register (ICR), while we simultaneously write to the keyboard control register to reissue the normal interrupt.

4.2.3 Universal Serial Bus Keyboard

Universal Serial Bus (USB) is a popular external interface standard that enables communication between the computer and other peripherals. There are currently three versions of USB in use: USB 1.1, 2.0, and 3.0. A USB system has a host controller, and it sits between the USB device and the operating system. USB 1.1 uses Universal Host Controller Interface (UHCI) [26]; USB 2.0 uses Enhanced Host Controller Interface (EHCI) [23]; and the recent USB 3.0 uses eXtensible Host Controller Interface (XHCI) [24]. From the manuals of these standards, all of the them support triggering SMIs. For instance, XHCI uses a 32-bit register to enable SMIs for every xHCI/USB event it needs to track, and we are able to trigger an SMI for every key press required by TrustLogin. This register is located at xHCI Extended Capabilities Pointer (XECP) + `0x04`, and we can find XECP from the base address of the XHCI + `0x10`. Similar registers that enable SMIs can also be found at EHCI and UHCI. Moreover, Schiffman and Kaplana [38] demonstrated that USB keyboards can generate SMIs.

4.3 Generating Placeholders

To replace the original password, we generate a placeholder for each keystroke intercepted in the SMI handler. One of the simplest methods is to replace each keystroke with a constant character (e.g., character ‘p’). However, this method cannot pass the security checks that ensure the strength of the password. For instance, most of the password policies require that passwords contain at least one digit, one lowercase character, one uppercase character, and one special character. Although these checks are usually performed on the server side, they could be done on the client application. To address this problem, TrustLogin replaces a keystroke based on its type. TrustLogin substitutes the original keystroke with a random one of the same type.

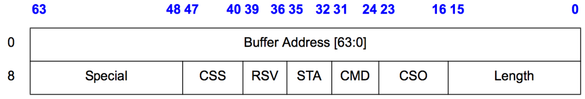
We use a linear-congruential algorithm to generate a pseudo-random number n in the SMI handler. The parameters of the linear-congruential algorithm we used are from Numerical Recipes [44]. Next, we use $n \bmod k$, where k is the cardinality of the corresponding type (e.g., 26 each for lower- or uppercase characters) to generate a random character. In terms of the special characters, different applications or servers may have a different set of valid special characters. For instance, the American Express website does not allow special characters like ‘.’ in the password, while Bank of America and CitiBank do accept it. TrustLogin assumes the application allows six special characters as follows: dot, underscore, star, percent, question mark, and sharp. We can always update the set of special characters based on the application requirements. Next, we discuss how the network card intercepts packets and replaces the placeholders with the original password.

4.4 Intercepting Network Packets

TrustLogin starts to intercept the network packets when the Enter key is received in the secure input mode. This means the user has finished entering the password and the OS is about to transmit the login credentials. We use a popular commercial PCI-based network card, Intel e1000 [25], to demonstrate this in TrustLogin.

Message Signaled Interrupts (MSIs) are an optional feature incorporated into PCI devices. They essentially allow a PCI device to generate an interrupt without having to make use of a physical interrupt pin on the connector. Introduced in PCI version 2.2, MSIs allow the device to send a variety of different interrupts to the CPU via the chipset. One such interrupt is the SMI. We can configure the MSI configuration registers (offset `0xF0` to `0xFF`) in the PCI configuration space to enable SMI triggering.

When MSIs are enabled, the network card generates a message when any of the unmasked bits in the Interrupt Cause Read register (ICR) are set to 1 [25]. The ICR contains all interrupt conditions for the network card. Each time an interrupt occurs, the corresponding interrupt bit is set in the register. The interrupts are enabled through Interrupt Mask Set/Read Register (IMS). For instance, the first bit of IMS sets a mask for Transmit Descriptor Written Back (TDWB). When the hardware finishes transmitting a packet, it sets a status bit back to the transmit descriptor; this action could be an interrupt condition. In TrustLogin, we reroute this interrupt to an SMI by using MSI. This means we can trigger an SMI for each packet when it is transmitted. In the SMI handler, we then inspect all of the transmit descriptors in the transmit queue and search for the login packet. It is possible that the first packet that generates the SMI is the login packet. To address this edge case, we create a transmit descriptor in the SMI handler beforehand and make sure the first SMI from NIC is triggered by this transmit descriptor. This transmit descriptor is created when TrustLogin enables the NIC’s SMI triggering by rerouting TDWB interrupts to an SMI. Note that the transmit queue may become empty before finding the login packet, so we may miss the first transmit descriptor that arrives at the empty transmit queue. Thus, we insert a transmit descriptor whenever the transmit queue becomes empty until identifying the login packet.



	63 48	47 40	39 36	35 32	31 24	23 16	15 0
0	Buffer Address [63:0]						
8	Special	CSS	RSV	STA	CMD	CSO	Length

Figure 3: Transmit Descriptor Format [25]

Figure 3 shows the format of the transmit descriptor structure. Buffer Address points to the data in the host memory. The CMD (i.e., command) field specifies the RS (i.e., report status) bit. With this bit set, the hardware writes a status bit back to the STA (i.e., status) field in the descriptor when a packet is transmitted. For the inserted packet, we set the RS bit and `NULL` to the Buffer Address so that it transfers no data. We also makes sure all of the inspected packets have the RS bit set.

To replace the placeholders in the network packets, TrustLogin can simply search the sequence of the placeholders in the packets. We can use the Transmit Descriptor Base Address (TDBA) and Transmit Descriptor Tail (TDT) to find the addresses of the transmit descriptors. The transmit descriptor structure contains all of the information about the packet including the address of the payload. Note that the addresses here are physical addresses (i.e., no paging) because the Direct Memory Access (DMA) engine of the NIC only understands the physical addresses. One challenge of this method is that the network packets are encrypted (e.g., TLS) and Section 5.3.1 discusses this further. After the SMI handler finds and replaces the placeholders, it waits until the packet leaves the host to avoid further sensitive data leakage. Moreover, the attacker may use NIC’s diagnostic registers to access transmitted packet connects. We empty the NIC’s internal Packet Buffer Memory (PBM) by writing 16KB random data since the size of the internal buffer of our testing NIC is 16KB [25].

4.5 Ensuring Trusted Path

One challenge of TrustLogin is the reliability of triggering SMIs. As shown in Figure 2, The I/O redirection table in red is not trusted. An attacker with ring 0 privilege can modify the table to intercept an SMI, and then prepare a fake switching process so that the users think that he or she is in the SMM. In this case, the attacker can trick the user and get the password. This is a typical spoofing attack. There has been some research tackling this problem [41, 34, 40, 32]. Bumpy [34] uses an external smartphone as the trusted monitor to acknowledge the switching. SecureSwitch [40] and Lockdown [41] use a dedicated switch to ensure the trust path. Cloud Terminal [32] uses a UI with strawberries on the screen as a shared secret to prevent spoofing attack. In

TrustLogin, we implement two novel methods to prevent the spoofing attacks. One approach is to use the keyboard Light Emitting Diode (LED) lights, and the other is to use the PC speaker. Next, we explain the implementation details of these two approaches.

4.5.1 Keyboard LED Lights

We use the LED lights on the keyboard to ensure the trust path. Usually, there are three LED lights on the keyboard, indicating Num, Caps, and Scroll locks. The users can set a shared secret LED light sequence to indicate that the system is in SMM. For instance, we can refer to scroll lock as 0, number lock as 1, and caps lock as 2. {[0 on]→[0 off]→[1 on]→[1 off]→[2 on]→[2 off]} is a LED light sequence. When the system switches into SMM, the SMI handler performs the shared secret LED light sequence so that the user knows the system is in SMM–not tricked by attackers.

To program the keyboard LED lights, we write a command byte, `0xED`, into the keyboard data register, and then write a LED state byte to the same I/O port. Bit 0 is for scroll lock; bit 1 is for number lock; bit 2 is for caps lock. Value 1 means on and 0 indicates off. Since every keystroke generates two interrupts (i.e., key press and release), TrustLogin only shows the LED light sequence when the key is released. We can easily identify a key release by checking the value of the scan code (greater than `0x80` [6]).

To help the user to identify the LED light sequence, we set a time delay between two lights. For instance, there should be a time delay between [0 off] and [1 on] for distinction. In TrustLogin, each light is on for 1 `ms`, and we set the same time delay when switching lights. The authors can identify that sequence based on their observations in the experiments. The user can adjust the time delay based on their preference.

```
  mov $0x30D40 , %ecx  ;1 CPU cycle
DELAY:
  nop                  ;1 CPU cycle
  nop                  ;1 CPU cycle
  nop                  ;1 CPU cycle
  loop DELAY           ;8 CPU cycles
```

Listing 1: Assembly Code that Introduces 1 `ms` Delay on Our Testbed

Listing 1 shows the assembly code that introduces a 1 `ms` delay on our testbed. This delay function loads a counter, `0x30D40` or 200,000 in decimal, into `EAX`, and spinlocks until the counter is 0. The value, 200,000, is calculated from the time it takes to execute the loop instructions on our testbed. The testbed has an AMD Sempron LE-1250 2.2 GHz processor with AMD K8 chipset. The `MOV` and `NOP` instructions take 1 CPU cycle and the `LOOP` instruction takes 8 CPU cycles [5]. We also assume it takes 7 CPU cycles for a `LOOP` instruction when the contents of `EAX` is zero. The equations explain the steps that calculates the counter for performing 1 `ms` time delay on our testbed.

$$TimeDelay = \frac{ClockCycles}{ClockSpeed}$$

$$1ms = \frac{1 + N * (1 + 1 + 1) + (N - 1) * 8 + 7}{2.2GHz} \Longrightarrow N = 200,000$$

4.5.2 PC Speaker

We also use the PC speaker to ensure the trusted path. TrustLogin plays simple music on the PC speaker when each key is pressed in the secure input mode. The users can choose their favorite melodies and embed them in the SMI handler. By recognizing their selected tone sequence, they can ensure that an SMI is triggered for their every key press. Thus, the selected music should be short but recognizable. TrustLogin plays a C major scale in the SMI handler to demonstrate this idea. Table 3 in Appendix shows the notes and corresponding frequencies for one complete octave starting from a middle C. We set the middle C as 523.25 Hz based on a musical reference guide [1].

To play a tone, we program the Intel 8253 Programmable Interval Timer (PIT) in the SMI handler to generate musical notes. The 8253 PIT performs timing and counting functions, and it exists in all x86 machines. In modern machines, it is included as part of the motherboard's southbridge. This timer has three counters (Counters 0, 1, and 2), and we use the third counter (Counter 2) to generate tones via the PC Speaker. We can generate different kinds of tones by adjusting the output frequency. The output frequency is calculated by loading a divisor into the 8253 PIT.

$$Divisor = IF/OF,$$

where IF is the input frequency of the 8253 PIT. IF used by the PIT chip is about 1.19 MHz, and OF is the output frequency. Column 3 of Table 3 shows the calculated divisors for the musical notes of an octave based on their output frequencies.

Playing a note on the PC speaker takes the following steps: 1) Configure mode/command register of the PIT chip through port `0x43` with value `0xB6`, which selects channel 2 to use and sets the mode to accept divisors; 2) load a divisor into channel 2 through port `0x42`; 3) turn on bit 0 and bit 1 of port `0x61` to enable the connection between PIT chip and the PC speaker; 4) set a time for the note to play; 5) turn off the PC speaker by configuring port `0x61`. Similar to the LED lights sequnce, we need to set a time delay so that the users can easily identify the music. In TrustLogin, each note is produced in 1 `ms`, and we set the same time of the delay between every two notes.

5. CASE STUDY

We study legacy and secure applications to demonstrate the effectiveness of TrustLogin. For the legacy applications that we referenced, they are normally built on a client-server architecture, and authentication occurs using a plaintext username/password pair. We consider Remote Shell (rsh), File Transfer Protocol (FTP), and Telent legacy applications. Note that TrustLogin is OS-agnostic for legacy applications because it does not need to reconstruct the semantics of OS kernels and rebuilds the packet in the NIC. For the secure applications that we noted, their network traffic securely encrypted. We consider Secure Shell (SSH), Secure File Transfer Protocol (SFTP), and Transport Layer Security (TLS) secure protocols.

5.1 Hardware and Software Specifications

We conduct the case study on a physical machine, which uses an ASUS M2V-MX_SE motherboard with an AMD K8 Northbridge and a VIA VT8237r Southbridge. It has a 2.2 GHz AMD LE- 1250 CPU and 2GB Kingston DDR2 RAM. We use a Dell PS/2 keyboard and PCI-based Intel 82541 Gigabit Ethernet Controller as the triggering devices. To program SMM, we use the open-source BIOS, Coreboot [19]. We also install Microsoft Windows 7 and CentOS 5.5 on this machine.

5.2 Case Study I: Legacy Applications

For legacy applications, we use FTP as the study example. Next, we demonstrate the effectiveness of our system on both Windows and Linux platforms. Figure 4 shows the screenshots of the FTP login with and without TrustLogin on Windows and Linux. We create an FTP account on a server. The username and password for the account is `hack3r` and `AsiaCCS.`, respectively. On Windows, we install the Free Keylogger Pro version 1.0 [4] and use the FTP client to connect to the server. We first start the keylogger to log keystrokes. Next, we login to the FTP server without TrustLogin enabled. As shown in subfigure 4(a), we can see that the keylogger records the timestamp, application name, username, and password in a file. The red rectangle shows the password is `AsiaCCS.` as recorded by the keylogger. Then, we enable TrustLogin and login to the FTP server again. However, the password recorded by the keylogger has been changed to a random string generated by TrustLogin. In other words, the keylogger cannot steal the real password when TrustLogin is enabled. We install Logkeys version 0.1.1a [9] on the CentOS 5.5, and subfigure 4(b) shows the results. Similar to the experiments on Windows, we login to the FTP server with and without TrustLogin enabled. We can see that the keylogger logs the random placeholders when TrustLogin enabled, and the keylogger cannot steal the login password. Note that attackers can easily steal the passwords from legacy applications by sniffing out the network. However, TrustLogin is used to address the general problem of securing keystrokes on the local host. Here studying the legacy applications emphasizes that TrustLogin is a framework that works with various applications to prevent keyloggers.

5.3 Case Study II: Secure Applications

TrustLogin replaces the password placeholders in a network packet when the packet is about to transmit. However, since most of the network packets are encrypted, we cannot simply search for the placeholder sequence though the encrypted data. For example, Transport Layer Security (TLS) encrypts the data of network connections in the application layer.

Secure Shell (SSH) is one of the most popular application protocols for access to shell accounts on Unix-based systems. There are several client authentication methods supported by SSH. For instance, we can use a public and private key pair to authenticate the client. However, we only consider the password-based authentication method in this paper. The password-based authentication method is the most commonly used authentication mechanisms in SSH. The SSH server simply uses a username and password to authenticate the client, and the password transmitted by the client to the server is encrypted by a session symmetric key. Unlike the legacy applications, we cannot simply search for the login packet and replace the original password. To address this problem, TrustLogin decrypts and re-encrypts the login packets using the session symmetric key. Next, we explain how we find the session states like the session key.

5.3.1 Session State Searching

As mentioned, SSH protocol uses a session symmetric key to encrypt the traffic. To decrypt the login packet, we first need to find the session states in the memory. Fortunately, SMM is able to access all of the physical memory because it has the highest privilege. TrustLogin uses signature-based searching. Typically, software data structures inevitably create some signatures in memory. For example, researchers extract SSL private key in memory by validating RSA/DSA structures on multiple applications including Apache, SSH, and OpenVPN [18].

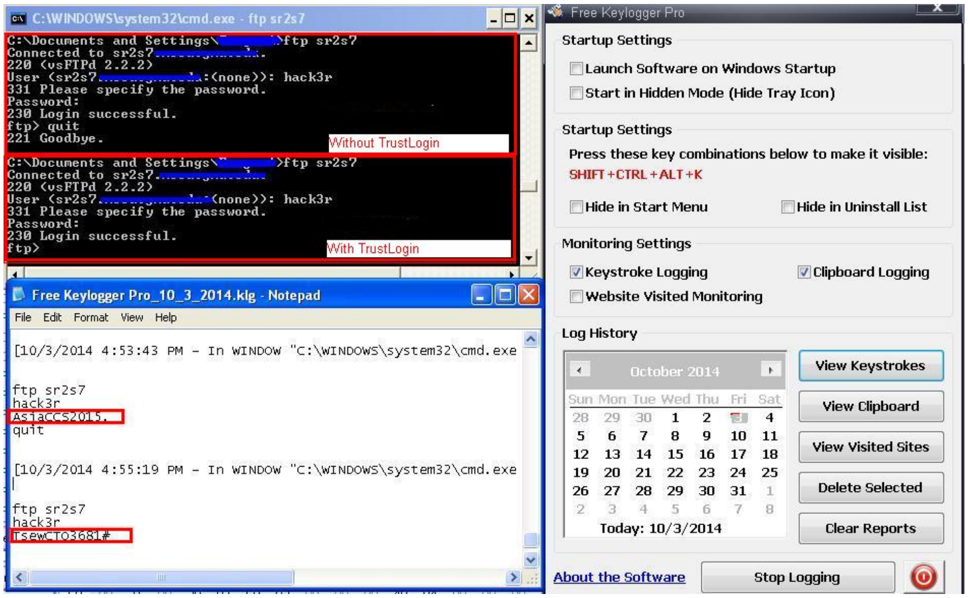
We use `session_state` structure as the signature for the searching. The `session_state` structure stores the session information for SSH communication. By analyzing the source code of OpenSSH [11], some fields of `session_state` structure in the `packet.c` file are static before users authenticate themselves. For instance, the `max_packet_size` field is set to constant, 0x8000. Listing 2 shows part of `session_state` structure, and these fields are static and continuous. Thus, TrustLogin uses this static signature as the search string and then finds the session states in memory. Since we assume malware has ring 0 privilege, an advanced malware can prepare a fake session key to TrustLogin. This attack breaks the login process but cannot steal the real password. Note that a DoS attack is out of the scope of this paper.

```
struct session_state {
  /* default maximum packet size, 0x8000 */
  u_int max_packet_size;
  /* Flag indicating whether this module has been
     initialized, 0x1 */
  int initialized;
  /* Set to true if connection is interactive, 0x0 */
  int interactive_mode;
  /* Set to true if we are the server side, 0x0 */
  int server_side;
  /* Set to true if we are authenticated, 0x0 */
  int after_authentication;
  /* Default before login, 0x0 */
  int keep_alive_timeouts;
  /* The maximum time that we will wait to send or
     receive a packet, -1, 0xffffffff */
  int packet_timeout_ms;
```

Listing 2: Static fields of Session Structure in SSH Source Code

One naive approach for implementation would be searching byte-by-byte through all of physical memory, but this is very slow. We instead search only the SSH instance in TrustLogin. Since only the physical memory is visible to SMM, we must reconstruct semantics of the raw data. In other words, we must understand the location of the SSH process and what its content means in memory. This is referred to as the "semantic gap problem." Recently, researchers proposed an array of approaches to address this problem [27], including hand-crafted data structure signatures [28, 47] and automated learning and bridging [30, 21]. Similar to the previous systems [28, 47], we manually reconstruct the semantics using kernel data structure signatures.

We first use the `ESP` value saved in the SMRAM to calculate the pointer to the current process's `task_struct`. Alternatively, we can obtain the address of the `init_task` from the `System.map` file. Next, we traverse the doubly linked list of `task_structs` or `run_lists` to locate the SSH process by comparing the `comm` field. Note that multiple instances of SSH could be running at the same time in the memory. We use the `prev` field for the transversal, which ensures that the first SSH process found is the last process launched. In this



```
C:\Documents and Settings\[redacted]>ftp sr2s7
Connected to sr2s7[redacted].
220 (vsFTPd 2.2.2)
User (sr2s7[redacted]:(none)): hack3r
331 Please specify the password.
Password:
230 Login successful.
ftp> quit
221 Goodbye.
```

```
C:\Documents and Settings\[redacted]>ftp sr2s7
Connected to sr2s7.[redacted]
220 (vsFTPd 2.2.2)
User (sr2s7.[redacted](none)): hack3r
331 Please specify the password.
Password:
230 Login successful.
ftp>
```

Free Keylogger Pro_10_3_2014.klg - Notepad
File Edit Format View Help

```
[10/3/2014 4:53:43 PM - In WINDOW "C:\WINDOWS\system32\cmd.exe


ftp sr2s7
hack3r
AsiaCCS2015.
quit


[10/3/2014 4:55:19 PM - In WINDOW "C:\WINDOWS\system32\cmd.exe


ftp sr2s7
hack3r
TsewCTO3681#
```

Startup Settings

□ Launch Software on Windows Startup
□ Start in Hidden Mode (Hide Tray Icon)

Startup Settings

Press these key combinations below to make it visible:
SHIFT+CTRL+ALT+K

□ Hide in Start Menu □ Hide in Uninstall List

Monitoring Settings

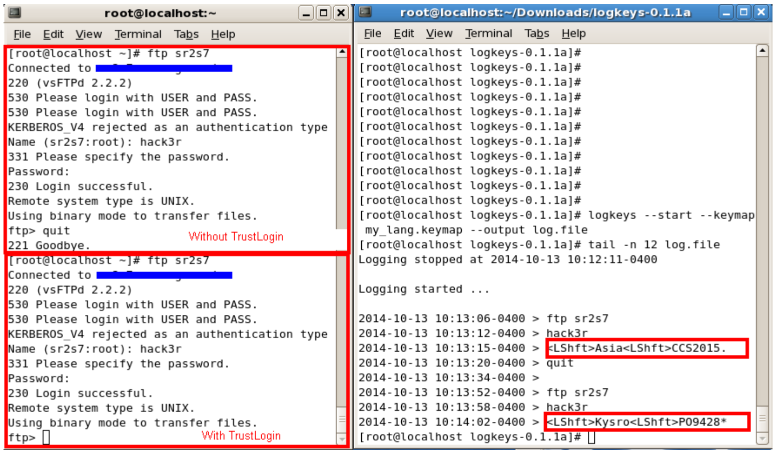
☑ Keystroke Logging ☑ Clipboard Logging
□ Website Visited Monitoring

Log History

Sun	Mon	Tue	Wed	Thu	Fri	Sat
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

Today: 10/3/2014

(a) FTP Login With and Without TrustLogin on Windows



```
[root@localhost ~]# ftp sr2s7
Connected to [redacted]
220 (vsFTPd 2.2.2)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (sr2s7:root): hack3r
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> quit
221 Goodbye.
```

```
[root@localhost ~]# ftp sr2s7
Connected to [redacted]
220 (vsFTPd 2.2.2)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (sr2s7:root): hack3r
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

root@localhost:~/Downloads/logkeys-0.1.1a
File Edit View Terminal Tabs Help

```
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]#
[root@localhost logkeys-0.1.1a]# logkeys --start --keymap
 my_lang.keymap --output log.file
[root@localhost logkeys-0.1.1a]# tail -n 12 log.file
Logging stopped at 2014-10-13 10:12:11-0400

Logging started ...

2014-10-13 10:13:06-0400 > ftp sr2s7
2014-10-13 10:13:12-0400 > hack3r
2014-10-13 10:13:15-0400 > <LShft>Asia<LShft>CCS2015.
2014-10-13 10:13:20-0400 > quit
2014-10-13 10:13:34-0400 >
2014-10-13 10:13:52-0400 > ftp sr2s7
2014-10-13 10:13:58-0400 > hack3r
2014-10-13 10:14:02-0400 > <LShft>Kysro<LShft>PO9428*
[root@localhost logkeys-0.1.1a]#
```

(b) FTP Login With and Without TrustLogin on Linux

Figure 4: FTP Login With and Without TrustLogin on Windows and Linux

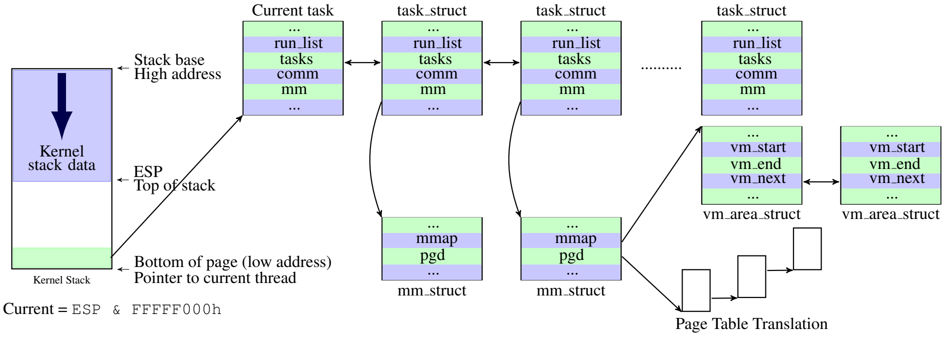


Figure 5: Filling the Semantic Gap by Using Kernel Data Structures in Linux

case, we assume the user interacts with the most recently launched SSH instance. Next, we obtain a pointer to the `mm_struct` from the `mm` field in `task_struct`. The `mmap` field in the `mm_struct` points to the head of the list of memory regions with the type `vm_area_struct`. The memory region object contains `vm_start` and `vm_end` fields, which define the start and end addresses of the memory region. Figure 5 shows the semantic reconstruction using kernel data structures in Linux. As pointed out in [27], all of the current solutions to the semantic gap assumes the kernel data structures are benign. Our semantic reconstruction approach also assumes this. We search the `session_state` signature in the memory regions of the SSH process, which achieves a better performance than the linear searching approach. Section 5.4 details the overheads of these two approaches.

All pointers in these structures are virtual. However, SMM does not use paging, meaning it addresses physical memory directly. Thus, we must translate addresses manually from virtual to physical space. For kernel-space structures (e.g., `task_struct` and `mm_struct`), there is a constant offset, `0xc0000000`, to move from virtual to physical space. For userspace structures (e.g., `vm_start` and `vm_end`), we locate and employ the process's page tables. Fortunately, the `pgd` field in the `mm_struct` stores the `cr3` value that tells us the location of the global page directory. After we retrieve all of the required information from memory, we decrypt the data and replace the placeholder sequence with the real password in the packet. Finally, we rebuild the network packet with the corresponding checksum by using the functions from OpenSSH source code.

5.4 Performance Evaluation

In order to understand the performance overhead of our system, we measure the runtime of each individual operation in the SMI handler. In TrustLogin, we have two parts of the handling code in the SMI handler; one is to handle the SMI triggered by the keyboard (i.e., KB SMI), and the other part is executed when the NIC triggers an SMI (i.e., NIC SMI). The KB SMI contains 7 steps.

1. Play a melody
2. Show an LED sequence
3. Disable keyboard SMIs (prevent reissuing)
4. Read the keystroke from the keyboard data register and save it in SMRAM
5. If Enter is pressed, break out and enable NIC SMIs
6. Generate random scan code and replace the keystroke
7. Enable keyboard SMIs for subsequent keystroke

Additionally, the NIC SMI consists of 3 operations.

1. Locate received packet in NIC memory
2. Search packets and inject original password
3. Disable NIC SMIs after password is injected

We measure the time delay for all of these operations in the SMI handler. As mentioned in Section 2, the hardware automatically saves and resumes the CPU context when switching to SMM. We also measure the overhead induced by switching to and resuming from SMM.

Table 1: Breakdown of the SMI Handler Runtime (Time: μs)

	Operations	**Mean**	**STD**	**95% CI**
KB SMI	Play music	26,244	3,675	[25,199,27,288]
	Show LED	6,317	251	[6,245,6,388]
	Disable KB SMI	1.47	0.21	[1.40,1.53]
	Read keystroke	2.38	0.33	[2.28,2.47]
	Enable NIC SMI	8.40	0.05	[8.39,8.419]
	Replace keystroke	8.94	1.27	[8.57,9.30]
	Enable KB SMI	1.14	0.17	[1.09,1.19]
	Total of KB SMI	32,583		
NIC SMI	Read NIC registers	3.96	0.10	[3.93,3.99]
	Search packets	18.27	1.18	[17.93,18.60]
	Disable NIC SMI	7.44	0.05	[7.42,7.45]
	Total of NIC SMI	29.67		
	Switch into SMM	3.29	0.08	[3.27,3.32]
	Resume from SMM	4.58	0.10	[4.55,4.61]
	Total of switching	7.87		

Table 2: Comparison between Linear Searching and Semantic Searching

Approaches	**Search Space**	**Time**
Linear Searching	2 GB	70.21 s
Semantic Searching	18 MB	(1.39+227.45) ms

Table 1 shows the time breakdown of each operation. We use the Time Stamp Counter (TSC) to measure the time delay for each operation. We first record the TSC value at the beginning and end of each operation. Next, we use the CPU frequency to divide the difference in the TSC register to compute how much time this operation. We conduct the experiment based on the FTP login for 30 trials. We calculate the mean, standard deviation, and 95% confidence interval for each operation. From Table 1 we can see that the total time for KB SMIs is about 32 ms. Note that most of the time is consumed by playing the melody and showing the LED sequence. Each note in the melody and each part of the LED sequence takes 1 ms (See Section 4.5 for details). The total time of the NIC SMI code and SMM switching are only about 30 and 8 μs, respectively.

As mentioned in Section 5.3.1, searching the SSH session states after reconstructing the memory semantic (called semantic searching) achieves a better performance than linear searching. To demonstrate this, we compare the linear searching with the semantic searching. We conduct the experiment on the Linux machine. We installed the latest OpenSSH client version 6.6p1 on the testbed, and the testing machine has 2GB physical memory. For linear searching, we compare the signature of the session states with the 2GB memory byte-by-byte. As for semantic searching, we first find the SSH process in memory using kernel data structures and then only search the memory regions pointed by `mmap`. After a user types `ssh username@hostname` in a terminal, the SSH server waits for the password from the client. At this time, we trigger an SMI and let the SMI handler perform both searching methods. We also use the TSC to measure the time it takes for each approach.

Table 2 shows the comparison between linear searching and semantic searching. The linear searching has 2 GB of searching space and takes about 70 seconds to find the session states. The semantic searching only has about 18 MB of searching space; it takes 1.39 ms to fill the semantic gap and 227 ms for searching.

6. LIMITATIONS AND DISCUSSION

TrustLogin provides a framework to secure password-based logins on commodity operating systems, and it can be applied to many network applications. We demonstrate the effectiveness by using legacy and secure applications. To port more secure applications working with TrustLogin, we need to decrypt and rebuild the relevant login packet, which requires application-specific efforts.

Additionally, an advanced keylogger may directly read the keyboard buffer to retrieve the keystrokes using DMA. For example, Ladakis et al. [29] proposed a GPU-based keylogger that directly reads the keyboard buffer from the GPU through DMA. In TrustLogin, the CPU has been switched to SMM, but DMA is able to read the keyboard buffer at the same time that SMM accesses it. In other words, there is a race between SMM and DMA. From the experimental results in Table 1 we can see that TrustLogin only takes a couple of microseconds (i.e., $3.29+2.38+8.94=14.61\ \mu s$) to switch into SMM and replace the scan code in the keyboard buffer. However, the GPU-based keylogger needs to periodically poll and monitor the keyboard buffer for new keystrokes, and the suggested polling interval is 90 milliseconds in the prototype of the GPU-based keylogger [29]. As stated in the paper, higher frequency of polling may affect the proper display of the graphics, and the user may notice the abnormal event. Thus, TrustLogin is able to defend a GPU-based keylogger as long as the polling interval does not pass below 15 microseconds.

There have been a number of attacks against SMM [16]. Wojtczuk and Rutkowska [46] have demonstrated a cache poisoning attack by configuring the Memory Type Range Registers (MTRR) to execute malicious code in the cache instead of SMRAM. However, this architectural bug has been fixed by the Intel in recent chipsets by adding System Management Range Register (SMRR). Additionally, the MITRE team discovered a buffer overflow vulnerability in SMM [17], but this bug is specific to the code implemented in the BIOS and is not an architectural bug. Recently, Wojtczuk and Kallenberg [45] presented an SMM attack by manipulating UEFI boot script. The UEFI boot script is a data structure interpreted by UEFI firmware during S3 resume. When the boot script executes, system registers like BIOS_CNTL (SPI flash write protection) or TSEG (SMM protection from DMA) are not set so that attackers can force an S3 sleep to take control of SMM. Fortunately, as stated in the paper [45], the BIOS update around the end of 2014 fixed this vulnerability.

7. RELATED WORK

Recently, researchers proposed several trustworthy computing environments to execute sensitive workloads. SICE [14] leverages SMM to provide an isolated execution environment to protect sensitive workloads in cloud servers. It does not rely on any software component in the host environment and supports up to 4GB of isolated memory. SecureSwitch [40] is another isolated computing environment for executing sensitive workloads. It relies on the S3 sleep mode (i.e., suspend to RAM) in the BIOS to switch between trusted and untrusted OSes. Flicker [33] uses Late Launch to enable execution of a Piece of Application Logic (PAL). Unlike these trustworthy computing environments, TrustLogin is lightweight and only protects the sensitive network inputs.

One of the closest works is Bumpy [34], a Flicker-based system for securing sensitive network input. It handles inputs in a special code module that is executed in an isolated environment using the Flicker. Bumpy can protect user's sensitive web inputs even with a compromised OS or web browser. However, Bumpy requires modification of the web application and web server, while TrustLogin does not and works transparently.

Cloud Terminal [32] is a micro-hypervisor and provides secure access to sensitive applications from an untrusted OS. It moves most application logic to a remote server called the Cloud Rendering Engine and only runs a light-weight Secure Thin Terminal on the end host, so end-users can securely execute sensitive applications. It also uses the Flicker to setup the micro-hypervisor. Cloud Terminal has the same threat model as TrustLogin, while TrustLogin uses existing hardware features without using a micro-hypervisor and has a smaller TCB.

Borders and Prakash proposed a Trusted Input Proxy (TIP) [15] to secure network inputs. The TIP runs as a module in a separate VM that proxies network connections of the primary VM. It also uses the placeholder approach to substitute the actual sensitive data. Neither TIP nor TrustLogin require modification of web browser and server, and they are transparent to users. As stated in the limitation section of this paper, TIP relies on a virtual machine monitor. It also introduces a large trusted code base and significant slowdown for I/O intensive applications. TrustLogin is a hardware-assisted method working on bare metal, which overcomes the shortcomings in TIP.

MP-Auth [31] uses a mobile device to encrypt the password under the public key of an intended server, and passes the encrypted password to the host for authentication. The host can only access a one-time password so that it can prevent keyloggers and phishing attacks. oPass [39] leverages a user's cellphone and short message service to thwart credential stealing and password reuse attacks. Recently, Google provided a 2-step verification for the Gmail login. The 2-step verification requires something you know (i.e., password) and something you have (i.e., cell phone). Even if an attacker compromises the password, she still needs the cell phone to get into the account, so the 2-step verification provides an extra layer of security. Compared to TrustLogin, MP-Auth, oPass, and 2-step verification propose a new protocol between the web browser and web server. That is, they modify the network application and server to fit their new protocols. TrustLogin is built on existing authentication protocols and is transparent to network applications and servers.

In recent years, an array of SMM-based systems have been implemented. On one hand, researchers use SMM to implement stealthy rookits [20, 38]. For instance, the National Security Agency (NSA) uses SMM to build an array of rootkits including DEITYBOUNCE for DELL and IRONCHEF for HP Proliant servers [10]. On the other hand, researchers leverage SMM to build security defense systems. For example, HyperCheck [49] and Hypersentry [13] are SMM-based systems used to check the integrity of the hypervisors. Spectre [47] and IOCheck [48] are introspection frameworks for detecting malicious malware in the live memory and firmware, respectively. SMMDumper [36] uses SMM to reliably acquire physical memory for forensics purposes.

8. CONCLUSIONS AND FUTURE WORKS

In this paper, we presented TrustLogin, a novel framework for securing password-login via System Management Mode. We do not trust underlying applications and OS on the target system, and TrustLogin is able to prevent rootkits and stealthy malware from stealing sensitive data from the local host. Since TrustLogin does not change any software on the client and server sides, it is transparent to users and applied applications. We implemented our framework on legacy and secure applications. The experiment results show that TrustLogin is robust against keyloggers on both Windows and Linux platforms. TrustLogin is able to defend spoofing attacks and guarantees the trust path of SMM switching. Our performance evaluation results show that TrustLogin is light-weight and only takes 8 microseconds for SMM switching.

As explained in the threat model, the current prototype of TrustLogin cannot defend against phishing attacks and only protects the login credentials on the local host, so we plan to mitigate this attack in our future work. For instance, we can ask users to type the destination hostame/IP address before entering the login credentials. Similar to intercepting the login credentials, we store the destination hostname/IP address in SMRAM. Note that this hostname/IP address cannot be modified by malware because we intercept the keystrokes before the OS receives them. When TrustLogin injects the login credentials back to the network packet, we also check the destination IP address of that packet and make sure it matches the one in SMRAM. If it matches, we can guarantee that the packet would be delivered to the user's desired destination. Note that the malware cannot change the IP address after the checking procedure because the SMI handler waits until the packet physically leaves NIC. Since humans are used to remembering hostnames instead of IP addresses, the SMI handler may need to implement an `nslookup` utility to translate the hostname to the IP address.

9. ACKNOWLEDGEMENTS

The authors would like to thank all of the reviewers for their valuable comments and suggestions. This work is supported by the National Science Foundation Grant No. CNS 1421747 and II-NEW 1205453, Defense Advanced Research Projects Agency Contract FA8650-11-C-7190 , and ONR Grant N00014-13-1-0088. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or the Navy.

10. REFERENCES

[1] C-Scale Frequency Reference Guide for Musicians. `http://www.ronelmm.com/tones/cscale.html`.

[2] Common Vulnerabilities and Exposures list. `http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html`. Access time: 07/06/2014.

[3] Credit Card Data Breach at Barnes & Noble Stores. `http://www.nytimes.com/2012/10/24/business/hackers-get-credit-data-at-barnes-noble.html?_r=3&`.

[4] Free Keylogger Pro. `http://freekeyloggerpro.com/`.

[5] Intel 64 and IA-32 Architectures Optimization Reference Manual. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[6] Keyboard Scan Code Set 1. `http://www.computer-engineering.org/ps2keyboard/scancodes1.html`.

[7] Keylogger Malware Found on UC Irvine Health Center Computers. http://www.scmagazine.com/keylogger-malware-found-on-three-uc-irvine-health-center-computers/article/347204/.

[8] Keylogger Products. `http://www.keylogger.org`.

[9] Logkeys Linux keylogger. `https://code.google.com/p/logkeys/`.

[10] NSA's ANT Division Catalog of Exploits for Nearly Every Major Software/Hardware/Firmware. `http://Leaksource.wordpress.com`.

[11] OpenSSH. `http://www.openssh.com`. Access time: 09/01/2014.

[12] Advanced Micro Devices, Inc. BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors. `http://support.amd.com/TechDocs/26094.PDF`.

[13] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[14] A. M. Azab, P. Ning, and X. Zhang. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[15] K. Borders and A. Prakash. Securing network input via a trusted input proxy. In *Proceedings of the 2nd USENIX workshop on Hot topics in security*, 2007.

[16] Y. Bulygin, J. Loucaides, A. Furtak, O. Bazhaniuk, and A. Matrosov. Summary of Attacks Against BIOS and Secure Boot. In *Defcon-22*, 2014.

[17] J. Butterworth, C. Kallenberg, and X. Kovah. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, 2013.

[18] N. Collignon. In-memory Extraction of SSL Private Keys. `http://c0decstuff.blogspot.com/2011/01/in-memory-extraction-of-ssl-private.html`, 2011.

[19] Coreboot. Open-Source BIOS. `http://www.coreboot.org/`.

[20] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: A New Breed of OS Independent Malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*, 2008.

[21] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[22] T. Holz, M. Engelberth, and F. Freiling. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *Proceedings of The 14th European Symposium on Research in Computer Security (ESORICS'09)*, 2009.

[23] Intel. Enhanced Host Controller Interface Specification for Universal Serial Bus. `http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ehci-specification-for-usb.pdf`.

[24] Intel. eXtensible Host Controller Interface for Universal Serial Bus (xHCI). `http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf`.

[25] Intel. PCI/PCI-X GbE Family of Controllers: Software Developer Manual. `http://www.intel.com/content/www/us/en/ethernet-controllers/pci-pci-x-family-gbe-controllers-software-dev-manual.html`.

[26] Intel. Universal Host Controller Interface (UHCI) Design Guide. `ftp.netbsd.org/pub/NetBSD/misc/blymn/uhci11d.pdf`.

[27] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. SoK: Introspections on Trust and the Semantic Gap. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[28] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-based Out-of-the-box Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[29] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *Proceedings of the European Workshop on System Security (EuroSec'13).*, 2013.

[30] T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P'11)*, 2011.

[31] M. Mannan and P. van Oorschot. Leveraging Personal Devices for Stronger Password Authentication from Untrusted Computers. *Journal of Computer Security*, 2011.

[32] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica. Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*, 2012.

[33] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[34] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *NDSS*, 2009.

[35] Ohloh. Black Duck Software, Inc. `http://www.ohloh.net`. Access time: 7/16/2014.

[36] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi. When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[37] S. Sagiroglu and G. Canbek. Keyloggers. *Technology and Society Magazine, IEEE*, 2009.

[38] J. Schiffman and D. Kaplan. The SMM Rootkit Revisited: Fun with USB. In *Proceedings of 9th International Conference on Availability, Reliability and Security (ARES'14)*, 2014.

[39] H.-M. Sun, Y.-H. Chen, and Y.-H. Lin. oPass: A User Authentication Protocol Resistant to Password Stealing and Password Reuse Attacks. *Information Forensics and Security, IEEE Transactions on*, 2012.

[40] K. Sun, J. Wang, F. Zhang, and A. Stavrou. SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS'12)*, 2012.

[41] A. Vasudevan, B. Parno, N. Qu, V. Gligor, and A. Perrig. Lockdown: A Safe and Practical Environment for Security Applications (CMU-CyLab-09-011). Technical report, 2009.

[42] VIA. VT8237R Southbridge. `http://www.via.com.tw/`.

[43] F. Wecherowski. A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. *Phrack Magazine*, 2009.

[44] H. William, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, New York, 2007.

[45] R. Wojtczuk and C. Kallenberg. Attacking UEFI Boot Script. `http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf`, 2014.

[46] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.

[47] F. Zhang, K. Leach, K. Sun, and A. Stavrou. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[48] F. Zhang, H. Wang, K. Leach, and A. Stavrou. A Framework to Secure Peripherals at Runtime. In *Proceedings of The 19th European Symposium on Research in Computer Security (ESORICS'14).*, 2014.

[49] F. Zhang, J. Wang, K. Sun, and A. Stavrou. HyperCheck: A Hardware-assisted Integrity Monitor. In *IEEE Transactions on Dependable and Secure Computing (TDSC'14)*, 2014.

APPENDIX

Table 3: Musical Notes of an Octave

Musical Note	Frequency (Hz)	Divisor
C	523.25	0x08E2
D	587.33	0x07EA
E	659.26	0x070D
F	783.99	0x06A8
G	783.99	0x05EE
A	880.00	0x0548
B	987.77	0x04B5
C	1046.50	0x0471