Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

1-2012

Virtualization Based Password Protection Against Malware In Untrusted Operating Systems

Yueqiang Cheng
Singapore Management University, yqcheng.2008@smu.edu.sg

Xuhua DING
Singapore Management University, xhding@smu.edu.sg

Follow this and additional works at: http://ink.library.smu.edu.sg/sis_research

Part of the OS and Networks Commons

Citation

Cheng, Yueqiang and DING, Xuhua. Virtualization Based Password Protection Against Malware In Untrusted Operating Systems. (2012). Research Collection School Of Information Systems.
Available at: http://ink.library.smu.edu.sg/sis_research/1670

This Conference Paper is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Virtualization Based Password Protection Against Malware In Untrusted Operating Systems

Yueqiang Cheng and Xuhua Ding

School of Information Systems,
Singapore Management University,
{yqcheng.2008,xhding}@smu.edu.sg

Abstract. Password based authentication remains as the mainstream user authentication method for most web servers, despite its known vulnerability to keylogger attacks. Most existing countermeasures are costly because they require a strong isolation of the browser and the operating system. In this paper, we propose KGuard, a password input protection system. Its security is based on the hardware-based virtualization without safeguarding the browser or OS. A security-conscious user can conveniently and securely activate or deactivate the password protection by using key combinations. We have implemented KGuard and experimented our prototype on Windows with Firefox. The results show that no significant performance loss is induced by our protection mechanism when a user authenticates to commercial web servers.

1 Introduction

Password based authentication is the primary method for a remote server to check a user's identity. In a typical web authentication, a user password is transferred from the keyboard to the kernel, then to the browser before being sent out over the network to the web server through an SSL channel. One of the main threats to password authentication is kernel/application keyloggers which steal the password from its transferring path.

Any countermeasure to keyloggers must cope with both the attacks on the application which forwards the password to a remote server, and the attacks on the I/O path, namely from the keyboard to the application. Virtualization based isolation is the main approach as used in [6, 9, 8, 3], where either the browser or the entire OS is isolated as a protected environment. This approach usually incurs significant cost due to the large code to isolate and the security assurance is not strong, though it addresses other related security problems, e.g., phishing attacks. Another approach, as suggested in Bumpy [16] and BitE [15], is to use an encryption-capable keyboard to protect the I/O path and rely on the latest processor features to isolate the application. However, most commodity platforms at present are not equipped with the needed keyboard.

In this paper, we propose a novel system to protect passwords against keyloggers in remote authentication without using a special keyboard or isolation like [6, 9, 8, 3]. Note that in the remote authentication setting, it is unnecessary for the user's platform (including the OS and the application) to know the *actual* password as long as it can forward the authentication information to the server properly. Therefore, the high level idea of our work is that a hypervisor intercepts the user's password input; and whenever the

application needs to submit the password to the server through an SSL channel, it traps to the hypervisor which performs the desired encryption. In other words, the normal SSL connection between the application and the server is split into non-cryptographic operations and cryptographic operations, such that the latter are accomplished by the hypervisor holding the password.

In our system, the cleartext password is never exposed to the operating system or the application. As a result, a keylogger can only get a ciphertext version. The system is highly efficient because no extra computation or communication cost is incurred as compared to normal password authentication, except the keyboard interception and the trapping. It is entirely transparent to the operating system, though the application needs to have a plug-in in order to split the SSL operations. Furthermore, the system is user friendly as it results in little user experience change. (Note that anti-phishing is not in the scope of our work.)

In the rest of the paper, we present the design and implementation details of our password protection system named as *KGuard*. It is for password based web authentication using Firefox. We also report its performance in experiments with commercial websites such as Gmail. A novel building block of our system is a secure user-hypervisor interaction channel that allows a user to authenticate a hypervisor, which in itself is of research value as it addresses one of the challenges recently identified in [30]. KGuard can be extended for other password authentication systems (e.g., SSH) by replacing the browser plugin with the one for the application.

ORGANIZATION. In the next section, we discuss the related work. Then we present an overview in Section 3 with the emphasis on the methodology used in our design. In Section 4, we describe the details of our design. The implementation details and performance results are shown in Section 5 and Section 6, respectively. We discuss several important issues in Section 7 and conclude this paper in Section 8.

2 Related Work

BitE [15] and Bumpy [16] are two isolation based systems that defend user input against malware attacks. Both of them require an encryption-capable keyboard. BitE suffers from a large TCB since it contains the legacy OS and Window Manager. Bumpy reduces the TCB size by using Flicker [14]. However, it has a higher computation latency. The KGuard system does not leverage the encryption-capable keyboard, and the TCB size of the KGuard is larger than Bumpy, and smaller than BitE.

Password protection against malware is a sub problem of password management which deals with other issues like phishing attacks. A widely used approach in [19, 8, 3] is to set up a secure compartment which functions as a proxy to help the user's authentication. For instance, TruWallet [8] and TruWalletM [3] use different techniques to secure the authentication proxy which securely stores the user credentials and properly submits them to a remote server. The main disadvantages of these schemes are the architectural change (e.g., GUI parts are required to moved from the legacy OS) on the platform and the high cost (e.g., longer data flow path comparing with the legacy one). In addition, it is challenging to isolate the browser and the user interface due to the enormous code size. Oprea et. al. in [19] propose an approach to allow users possess-

ing a trusted mobile device (e.g., PDA) to delegate their credentials (e.g., password) for performing a task (e.g., login). Other works in password management include PwdHash [22] which uses cryptographic techniques to cope with phishing attacks, and WebWallet [29] which checks user information submission and determines phishing attacks. Note that most secure password management systems are complementary to our work which focuses on password input.

Our work is also related to I/O protection in the kernel space. DriverGuard system [5] provides a generic solution to protect the confidentiality of I/O data from being attacked by a compromised kernel. However, it does not solve the password protection problem because it cannot protect the password residing in the application.

3 Overview

This section presents an overview of our work. We explain the design criteria and the rationale we follow, including the trust model and a high level explanation of our approach. We also show the architecture of the proposed system.

3.1 Design Criteria

Ideally, a password protection system should meet the following criteria. Firstly, the protection should offer the strongest security assurance. It should be able to defeat attacks from rootkits which subvert the operating system, as kernel rootkit keyloggers are not uncommon in the cyberspace. From the practicability perspective, the protection should induce little or no modification on the operating system and is fully compatible with existing browsers. This is due to the fact that proprietary operating systems such as Windows and Mac OS are more widely used than open-source operating systems. Furthermore, the password security should not be attained at the price of the easy-of-use of password authentication. On the user side, the protection scheme should be as simple as possible and does not require user possession of extra devices, such as a USB token and a mobile phone. On the server side, no changes should be needed. Last but not the least, the protection system should incur low cost. The cost is measured in terms of both the time delay during the password authentication session and the overall computation load on the platform. It is crucial that the user should not experience noticeable delay in an authentication session.

3.2 Design Rationale

In order to meet the criteria, we carefully assess a variety of design options. The foremost issue to consider is the trust model, i.e. which component in the platform can be considered as trustworthy.

Trust Model We do not trust the operating system and applications running on top of it, in the sense that they can be compromised and attempt to steal user passwords. Therefore, safeguarding user password necessitates a root of trust which should not

be subverted by rootkits. One candidate for the root of trust is the TPM chip [28], which is expected to resist all software attacks. Nonetheless, despite of its high security assurance, the TPM chip offers rather primitive and inflexible functionalities and is slow in computation. These drawbacks make it ill-suited for password protection.

In this work, we choose the hypervisor (a.k.a. virtual machine monitor or VMM) as the root of trust, as in [24, 25, 4]. The main benefit is that it allows us to develop desirable protection functions within the hypervisor, and therefore facilitates the design and the implementation. The hypervisor is not as secure as the TPM chip since several attacks have been discovered to compromise some versions of hypervisors [27, 7, 11, 21]. However, the security of the hypervisor can be ensured by three measures. Our design is based on hardware-assisted virtualization, such as Intel VT-x and AMD V, which significantly reduces the virtualization code of the hypervisor. In addition, TPM-based authenticated bootup can verify the integrity of the hypervisor when being launched. Thirdly, the hypervisor in our system is only for protection in a normal personal desktop setting, rather than a cloud server with a full-fledged virtualization for multiple VMs. Therefore, those unneeded services from the hypervisor are turned off so that only a minimal attack surface is exposed to the guest OS.

A secure hypervisor is capable to dynamically protect memory regions and I/O ports against direct malware accesses. In addition to that, the hypervisor also uses IOMMU to enforce the similar policies against malicious DMA operations launched by malware.

Protection Method There exist several candidate methods to protect user passwords against rootkits. One is to follow the isolation approach as shown in [13]. The execution of routines processing the password is isolated from the rest of the platform to cordon off attacks. This method is not compatible with our design criteria because of its low performance. The frequent interrupt caused by user keystrokes for password inputting induces the expensive system thrashing between the protection mode and the regular mode. In addition, the isolation approach faces the difficulty of extracting appropriate Pieces of Application Logic (PAL) due to the complexity of the kernel's keyboard input processing and the browser's web page processing. Another possible method could be to escort the password data flow as shown in DriverGuard [5]. Nonetheless, this approach requires code modifications on the drivers, which does not satisfy our compatibility requirement. Moreover, DriverGuard by itself does not guarantee the security of password in the application level.

In this work, our method is based on the characteristics of the password authentication. Firstly, passwords are typically sent to a remote server through an SSL/TLS connection. It is *not* necessary for the local host to know the password in use. Secondly, passwords are fed to a system through keystrokes which can be intercepted by the hypervisor.

Based on these two observations, the basic idea of our protection method is to intercept the password keystrokes and then securely inject them back to the SSL/TLS connection established by the browser, however, with its cryptographic operations performed by the hypervisor. Therefore, the password is encapsulated using the web server's public key following the SSL/TLS specification without any exposure to the operating system or the browser.

Security Properties The main challenge of realizing the proposed protection method is the gap between the hypervisor and the security-conscious user. In existing platforms, a user only interfaces with the operating system through the application, e.g., a browser.

This gap entails three problems to solve. The first is about the timing for protection. It is undesirable for the hypervisor to intervene in all keyboard inputs. Ideally, the protection is only activated by the user whenever needed. The *on-demand* protection brings up the second challenge: how the user is assured that the hypervisor is protecting the password input. Note that the operating system may cheat the user by simulating the hypervisor's behavior. Last but not the least, the hypervisor's SSL traffic assembling must use a proper public key certificate for encapsulation. Ideally, the hypervisor is capable of verifying whether the certificate belongs to the intended web server.

In this work, we design a dynamic secure channel for user-hypervisor interaction which bypasses the operating system. While the hypervisor's protection mechanism is dormant, the channel allows a security-conscious user to activate it through a key combination. In addition, the channel allows the user to verify whether it is indeed active. Note that it is not necessary for the hypervisor to authenticate the origin of the keystrokes, because a faked activation key combination, e.g., from the malware instead of the user, does not lead to password leakage[1].

For the aforementioned third problem, our design achieves the same level of security as the standard browser's dealing with SSL certificates, because a certificate misuse is essentially the traditional man-in-the-middle attack on SSL. Similar to the browser's certificate verification, the hypervisor ensures that the certificate is genuine and matches the SSL connection.

3.3 The Architecture

We consider a platform with an operating system running on top of a hypervisor. A user uses a web browser to login to a remote server by supplying the password. KGuard is designed to protect the user password from being stolen by kernel/application rootkits. The architecture of KGuard consists of three components:

1. A secure user-hypervisor interaction channel allows the user to activate or deactivate the password protection and authenticate the hypervisor. A user toggles the protection by pressing a prescribed key combination. In response, the hypervisor securely displays (on the screen) a secret message pre-shared with the user.
2. A routine in the hypervisor intercepts user keystrokes after the protection is activated. It also validates the authentication server's public key certificate supplied by the browser and encapsulates the password using encryption.
3. A browser plugin splits the SSL connection for password submission. Specifically, it requests the hypervisor to perform the needed cryptographic operations in a SSL connection and handles other non-cryptographic operations by itself.

[1] The faked activation key combination can be considered as a denial of service attack. It will be quickly spotted by a user because as shown later, the hypervisor will respond to the user with a secret message pre-shared with the user.

Note that the hypervisor only performs cryptographic operations. It does *not* establish any SSL connection with the server. In a web authentication, the browser may establish multiple SSL connections. Only the one submitting the password is split by the plugin to get the needed cryptograms from the hypervisor. The benefit of this design is that it does not entail extra computation and communication cost and it can keep the hypervisor small without including the support for SSL.

4 The Design Details

4.1 User-Hypervisor Interaction

The user-hypervisor interaction channel is a duplex channel. In one direction, a user sends an activation command to the hypervisor by requesting the operating system to issue a hypercall. In the other direction, the hypervisor (on receiving the user's command) securely displays a secret message on the screen. Therefore, the user can verify whether the hypervisor receives the command or not.

Hypervisor Protection Activation There exist several approaches for activation. One alternative design is for the hypervisor to listen to a prescribed hardware event, such as keystrokes, plugging a USB device etc. These methods can bypass the operating system. Nevertheless, it requires extra work from the hypervisor which has to keep listening to all events and filter them properly. In our system, we do not favor this approach because 1) we aim to minimize the load on the hypervisor, especially when the protection is not needed; and 2) bypassing the operating system is not necessary because no *data* is sent to the hypervisor for activation. In addition, the user can verify the activation by checking the returned secret message from the hypervisor.

In our design, the operation system is the medium transferring the user's activation command to the hypervisor. Specifically, we design an application routine, e.g. a browser extension, and install a new module to the OS, e.g. a virtual device in Windows. The application routine listens to a prescribed key combination (i.e., the activation command). When the event is captured, it issues a hypercall to inform the hypervisor. Specifically, in the system initialization phase, the hypervisor prepares a hypercall table and then the installed OS module maps the table into the kernel space. The module exports an interface (i.e., a system call) to applications. After getting input parameters from an application via the exported system call, the module is invoked and forwards these parameters to the hypervisor through a hypercall as the Xen Hypercall mechanism [1].

In response to the activation hypercall, the hypervisor clears the keyboard input buffer, starts to intercept the keyboard strokes as described in Section 4.2, and authenticates itself to the user as shown in the next subsection.

Visual Verification of Hypervisor Protection The verification of hypervisor protection requires an output interface. To ensure its security, the output should not be captured or manipulated by malware in the guest OS. Otherwise, the guest can impersonate the hypervisor and give the user an illusion that the protection is activated.

The basic idea of our visual verification is that the hypervisor securely outputs to the monitor a secret text message priorly chosen by the user. Note that without involving the operating system, the monitor automatically and periodically fetches the display data *directly* from a memory region called the *display buffer*, whose location is determined by the hardware [10], and then it renders them on the screen. The hypervisor shows the secret message to the user by writing it into the display buffer. To prevent the operating system from attacking the secret, the hypervisor clears the _PAGE_PRESENT attribute bit of the corresponding page table entries. As a result, any guest access will be denied by the hardware.

The details of the visual verification are described below. Initially, the user chooses a random text message as his/her long term secret shared with the hypervisor. When the hypervisor boots up, the secret message is passed to the hypervisor as a booting parameter, which is the reason why the secret has to be text. Once taking control, the hypervisor stores the secret message into its own space. Since the hypervisor boots up before the operating system, the OS is not able to access this secret. To display it on a monitor in the graphics mode, the hypervisor derives the graphic version of the secret message by using the corresponding font bitmap for each character.

After receiving the activation hypercall, the hypervisor substitutes a part of the display buffer with the secret graphic data. As a result, the user secret message is displayed on the screen. The location of the message on the screen depends on its offset in the display buffer. Note that it is unnecessary to choose random locations. In addition, the hypervisor properly sets the attribute bits of the page table entries covering the graphic secret. Secret uploading and attribute bit setting up are an atomic operation. In other words, the hypervisor occupies the CPU without yielding it to the operating system until the attributes are set.

The hypervisor then sets up a timer whose duration is configured by the user during bootup. When the timer expires, the hypervisor restores the original display data, and finally returns the page access rights back to the guest OS.

Hypervisor Protection Deactivation Protection deactivation requires a stronger authentication on the user than protection activation, since malware may attempt to impersonate the user to terminate the protection. Note that once the protection is activated, the hypervisor has cleared all previous data in the keyboard input buffer and intercepts all new keystrokes. As a result of the interception, no software can access the keyboard input buffer, either directly or through DMA operations, as explained in Section 4.2. Only the physical keyboard strokes can place inputs to the buffer.

Therefore, the hypervisor in KGuard is pre-configured with a deactivation command. Once it intercepts the command during its protection, it switches to the no-protection state by releasing the access control on the keyboard input buffer.

4.2 Keystroke Interception

After getting the activation key-combination command from the user, the hypervisor starts keystroke interception. Since the key stroke code is directly delivered to the guest's memory by the hardware using DMA, keystroke interception means that the hypervisor retrieves the keyboard scan code *before* the guest.

One potential approach is for the hypervisor to intercept all interrupts and intervenes if needed. The main drawbacks of this approach are twofold. This approach may fail because the guest OS can keep scanning the keyboard input buffer without waiting for the interrupt. Therefore, the guest OS may have the luck of getting the data prior to the interrupt. Secondly, the interrupt number can be shared by several devices. The hypervisor has to determine whether the interrupt is for the keyboard. Furthermore, the interrupt by itself does not provide sufficient information for the hypervisor to locate the data.

Since locating the keyboard input buffer is an indispensable step, we let the hypervisor intercept the guest access on the keyboard input buffer, rather than interrupt interception. This method reduces the burden of the hypervisor as the guest OS manages all interrupts and is forced by the hardware to alert the hypervisor for the scan code retrieval. For this purpose, the hypervisor sets up page-table based access control on both the keyboard I/O control region storing I/O commands and the keyboard input buffer storing the scan code. IOMMU is also configured such that no DMA command can be issued to access these protected regions. Consequently, both the guest OS's keyboard I/O command issuance and its data retrieval are intercepted by KGuard. For the I/O control, KGuard emulates the operations; for the data retrieval, it replaces the user keystroke with a dummy one and saves the original input into a buffer in the hypervisor space.

The actual access control mechanism for the keyboard input buffer depends on the keyboard interface. A PS/2 keyboard usually uses PIO to transfer data whereas a USB-keyboard uses DMA. It is easy to deal with port I/O keyboards. The technique for controlling I/O port has been demonstrated in [5]. The access control for USB-keyboard is more complex due to the USB architecture. The so-called *Universal Host Controller* hardware uses a 32-bit register called FLBASEADD to locate a list of *frame pointers*. A frame pointer points to a list of *Transfer Descriptors (TDs)*. A TD specifies the necessary I/O parameters for one DMA operation, including the input buffer address. After completing one keyboard I/O, the guest OS must either update the current TD or insert a new TD in order to read the next keyboard input. The keystroke interception for a USB keyboard follows the steps below.

Step 1. KGuard freezes the present frame list and all TDs by setting FLBASEADD and all memory regions occupied by the frame list data structure as *read-only* using I/O bitmap and page table respectively. Therefore, any attempts from the guest OS to relocate the input buffer will be monitored by KGuard.

Step 2. KGuard locates the keyboard input buffer following the path used by the host controller. The keyboard input buffer is then set as *inaccessible*.

Step 3. When the guest OS attempts to read the keyboard input buffer, a page-fault is generated and passes the control to KGuard which saves the scan code (which is one password character) in the input buffer and replaces it with a dummy one, and sets the buffer as *read-write*. The guest OS can have a full access to this buffer.

Step 4. When the guest OS prepares for the next keyboard I/O by updating the TD, a page-fault is generated. In response, KGuard emulates the update operation. To prevent malware from providing faked keystrokes, the hypervisor clears the content

in the keyboard input buffer, which ensures that the data fetched in Step 3 is indeed from the keyboard.

Note that KGuard responds differently on the keyboard input buffer and the I/O region because one keyboard I/O only involves one TD update but may incur multiple accesses to the buffer depending on the driver's needs. Our approach avoids unnecessary hypervisor involvements.

We further remark that the keyboard interception is only activated based on the user's command. With the cooperation from the user, the incurred cost is therefore minimal to the platform's overall performance and it is reasonable for KGuard to treat all the intercepted keystrokes as the password. Even in case that the user and KGuard are out of synchronization, no user secret is compromised and the user can easily reset the protection.

4.3 Handling SSL Session

A normal web authentication may involve one SSL session comprising one or multiple SSL connections. Typically, when the user clicks a button for password submission, the browser sends out the encrypted password with other necessary information through an SSL connection.

In our system, the browser is deprived of the privilege of handling the password, because the encryption of the password and other authentication information must be performed in the hypervisor space, instead of in the untrusted guest domain. For this purpose, we design a dedicated browser extension for posting authentication information to the server through SSL. To achieve both security and compatibility, the extension is only responsible for non-critical operations in the SSL connection, while all cryptographic operations, such as master key generation and data encryption, are exported to KGuard.

The extension captures the login event and initiates a new SSL connection with the server. All keys used in this SSL connection are *newly* derived and only known by KGuard and the server. Note that this new connection will be immediately closed after the login event. Therefore, the browser does not need to maintain any extra connection. In the new SSL connection, the extension obtains the server's public key certificate. At the same time, it prepares a data blob containing all the data needed by the web server (except the password), e.g., the user name. It then submits to the hypervisor the data blob together with the server certificate. The hypervisor merges the blob with the intercepted user password, and encrypts them following the SSL specifications, on the condition that the provided public key certificate is valid. On receiving the resulting ciphertext from the hypervisor, the extension prepares the SSL data and sends them to the server. If the authentication succeeds, the server usually returns a URL with some cookies, which are decrypted by the hypervisor and forwarded to the extension. The extension then sets the cookies and redirects the browser to the URL. Now the extension terminates its SSL connection. Since neither the extension nor the browser possesses the keys for the SSL connection used for password submission, this SSL connection cannot be reused by the browser.

To avoid verbosity, we do not recite how the hypervisor generates the master key and performs the encryption, because it strictly follows the SSL/TLS specification. Out of the same reason, we do not explain how the extension prepares the data blob and the SSL traffic. However, it is worthwhile to elaborate how the server's public key certificate is validated by the hypervisor. Since we do not trust any software in the guest domain, the certificate forward by the extension to the hypervisor can be a malicious one. If the adversary has the corresponding private key, the hypervisor's password encryption will be decrypted by the adversary. We leave the details of the browser extension in Section 5 because it is browser specific and more relevant to usability than security.

Server Certificate Verification Certificate verification has long been considered as a thorny problem due to the trust on the public key infrastructure. The problem is even more complicated in our case because limited information is provided to the hypervisor for the sake of minimizing the hypervisor's size. Note that phishing detection is *not* within the scope of our study. Therefore, the criterion of a certificate's validity is not whether it matches the web server the user intends to login. Instead, a certificate is deemed as trusted as long as its root CA is trusted by the user.

In our system, the user may choose to trust all pre-loaded root CA certificates or import CA certificates she trusts. Once the user obtains a repository of trusted (root) certificates, the crux of our system is how the user securely passes them to the hypervisor. The difficulty is that the hypervisor does not have a file system and the whole guest is not trusted. The solution we propose relies on an additional trusted platform, or alternatively, the user may consider his/her platform in the initial state is trustworthy. On such a trusted platform, cryptographic tools such as OpenSSL, can be used to compute a HMAC key H_k and computes HMACs for each of the trusted certificate. Then, the user imports all trusted certificates as well as their corresponding HMAC tags into a file on the untrusted platform running with KGuard. During the platform's rebooting, the HMAC key H_k is passed to the hypervisor as a parameter. Therefore, the hypervisor knows whether a certificate is trusted by the user by checking its HMAC tag. Instead of using HMAC, the user may also apply digital signatures and pass the public key to the hypervisor, though this approach is not preferred because of its longer key and higher computation cost. Note that these above procedure is only executed *once*, i.e. for the first time using KGuard. All HMAC tags in the file are able to be reused after rebooting.

In runtime, the certificate verification proceeds as follows.

Step 1. The browser extension receives the public key certificate from the server and composes a certificate chain such that the last certificate in the chain is a trusted certificate imported by the user. For ease of description, we denote the certificate chain as $(Cert_0, \cdots, Cert_k)$ where $Cert_0$ is the server's certificate and $Cert_i$ is the issuer of $Cert_{i-1}$ for $1 \leq i \leq k$. In most cases in practice, $k = 1$ or 2. Note that only $Cert_k$ is the trusted certificate while all others are not. It is not necessary to obtain the issuer for $Cert_k$ even if it is not a root, because it is already trusted.

Step 2. The extension transfers $(Cert_0, \cdots, Cert_k, \sigma)$ to KGuard, where σ is the HMAC tag for $Cert_k$. In addition, the extension transfers the server's host name to KGuard. The transferring is accomplished by a hypercall.

Step 3. In response, KGuard first checks whether σ is a valid HMAC for $Cert_k$ using the HMAC key provided by the user during bootup. If the checking fails, KGuard rejects the certificate chain and aborts.

Step 4. KGuard then verifies the certificate chain in the same ways as the browser's verification, by treating $Cert_k$ as a trusted CA. Namely, it checks $Cert_i$'s signatures with the public key in $Cert_{i+1}$ for $0 \leq i \leq k-1$, and make sure that they are not expired, and checks whether $Cert_0$'s subject name matches the given server hostname (domain name). If all certificates pass the checking, KGuard accepts $Cert_0$ as the server's public key and uses it to encrypt the pre-master secret key in the current SSL connection.

The hypervisor calculates an HMAC value of each certificate in the verified certification chain, and returns them back to the guest if the certificate chain passes all checks. The browser inserts the certificate with its HMAC tag into the trusted certificate repository. This is to save the hypervisor's verification time when this certificate is reused in the user's future logins. Note that the new website certificates are accepted once the root certificate is trusted by the user.

4.4 Security Analysis

The security of the proposed password protection mechanism relies on the security of the hypervisor and the user cooperation. With the assumption on both conditions, the user-hypervisor channel ensures that the password is typed in only when KGuard is in position for keystroke interception, which saves the real password in the hypervisor space. The hypervisor and the guest space isolation enabled by the virtualization techniques prevents the guest from accessing the password. When the browser runs an SSL connection to submit the password, all cryptographic operations are performed by the hypervisor. The browser and the guest OS only get the ciphertext of the password. The hypervisor security is discussed in the Section 7.

5 Implementation

5.1 KGuard in The Hypervisor

We have built a prototype of KGuard on Xen 4.1.0 on a desktop with an Intel(R) Core(TM) i7 CPU-860 @2.80GHz processor and 4GB main memory. We choose a USB-keyboard as the experiment device. The implementation of KGuard does not depend on the design of Xen and can be easily migrated to other hypervisors.

KGuard consists of around 1500 SLOC for its main functions except cryptographic functions. We import the needed crypto functions (about 5000 SLOC) from [23]. The main cost is due to AES and RSA algorithms which need about 3500 SLOC. Nonetheless, comparing with the Xen code base (around 225,000 SLOC), we only increase the code size 2.885%. In fact, most of the code in Xen are not used by our system. Therefore, it is one of our future work to customize Xen for KGuard.

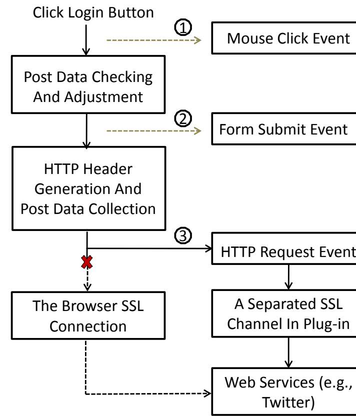

Fig. 1. Firefox events in the login session where the third one is intercepted by the extension

Visual Verification One of the implementation issues about the user's visual verification of the hypervisor verification is to choose a proper secret message. It is similar to a password in the sense that it should not be random enough to resist dictionary attacks, and it should be easy to remember. Since the user does not type in the message at runtime, the message can be much longer than a password. For instance, we choose the string "$ApBlE@8s_BaeuTifu10O$" as the user secret in our experiment.

Another issue is the position of the text message on the screen. We do not change the position for two reasons. Firstly, it does not enhance the security. If malware can breach the access control, it may grab the entire display buffer data. Secondly, from the usability perspective, it is inconvenient for users to find the message over the whole screen. We choose the top-left corner of the screen as the location because it is less likely to be overlapped with the web page in use.

The third concerns in visual verification is the performance overhead due to the slow speed of the display memory. It requires twice display memory access for the hypervisor to save the present content and to write the secret message. In our implementation, we use the following trick to save one display memory access. We do not save the original data. Instead, we impose the font bitmap of characters in the message upon the existing content. By performing the XOR operation, all the bits corresponding to the characters are flipped. As a result, the shape of the character is displayed on the screen. Although the content is not saved, it can be recovered by running the XOR operations again.

Note that our current implementation requires to work with the VGA compatible graphics cards.

5.2 Browser Extension and Plugin

Benefiting from the virtualization features of the Intel processor, we launch a hardware virtual machine (HVM) running Windows. The HVM guest domain runs a installation of Windows 7 Professional version with default configuration. We choose the popular firefox (version 3.6) as the test browser, and extend it with a plug-in and an extension.

The main part of the browser plug-in is based on CyaSSL v2[2]. It interacts with the hypervisor using hypercalls to build a separated SSL channel with a web server. Specifically, The plug-in interacts the hypervisor in the SSL handshake phase for four times: to transfer the server certificate chain; to provide the key materials for pre-master key generation; to provide the authentication data for encryption; and to provide a finish-message to terminate the SSL handshake phase. The plugin finishes the SSL protocol and forwards the server response data to the browser extension.

The browser extension is implemented using Firefox XML User interface Language (XUL) and JavaScript. One of the tasks of the extension is to listening to the user activation key combination and then sends a hypercall to KGuard. The other two tasks are to integrate the password protection with the browser. The first task is to intercept the authentication data submitted to the server. Since KGuard is transparent to the browser, it proceeds as usual in password submission though with a dummy password.

The events generated by Firefox after the login button is clicked are shown in Figure 1. We choose to intercept the *HTTP Request Event*, the last event right before Firefox is about to pass the data to the SSL layer. The benefit of this choice is that this event implies that the browser has prepared all the data (including the HTTP header) expected by the web server. Therefore, the extension does not need to handle the nuisance of gathering all kinds of POST data required by the web server.

The second task is to navigate the browser to the destination URL that is in the server response packages. After receiving the response packages returned by the plug-in from its own SSL channel, the extension extracts the cookies and the redirection URL by parsing the header and body. It updates the cookies in the browser, and requests it to refresh the current page to the redirection URL. For the following connections, no matter whether they are HTTPS or HTTP connections, the browser will send the request with corresponding cookies, and continue the web session as normal. Note that the browser is not aware of the existence of the separated SSL connection, thanks to the statelessness of HTTP and HTTPS protocols.

5.3 Hypercall Support In the HVM

In the Windows kernel space, we build a virtual device module using the Windows Driver Kit (WDK) [17]. The module first uses the instruction *CPUID* to find registers that contain the size and the location of the hypercall table. Then it maps the hypercall table into its own memory space. Using the mapped hypercall table, the module is able to issue hypercalls to communicate with the hypervisor.

The module also exports a *DeviceIOControl* interface for application usage. According to the $dwIoControlCode$ parameter in the *DeviceIOControl* interface, the module can request different services by issuing different types of hypercalls to the hypervisor.

[2] CyaSSL is a C-Langue SSL library for embedded and realtime operating systems, and in regular desktop and enterprise environments [12]

6 Performance Evaluation

We have run experiments and evaluated the performance and usability with legitimate web servers, including Google, Groupon, Twitter and Amazon, and Microsoft Hotmail. We divide the total authentication session into two phases to facilitate the evaluation. The first phase is user password input and the second is password submission. We have measured the time overhead in each of them. Note that our protection is the "on-demand" mode, therefore, there are no extra cost for the system when the protection is inactive.

6.1 Overhead for Password Input

Table 1 lists the time costs for the procedures taking place during a user's password inputting. The password input phase begins with protection activation and ends with protection deactivation. The main overhead is due to the hypervisor's responses to the activation/deactivation command and its interception of keyboard strokes. The activation cost mainly includes a guest system call, a hypercall, a series of access control setup, and two accesses on the display memory. The deactivation cost only includes the removal of access control on the relevant regions. The keystroke-interception cost is the CPU time spent for intercepting one keystroke. It includes two exceptions, emulation of the refreshing of TD and processing the keystroke.

Note that the user secret message is written to the display memory, instead of the main memory. Its speed is only 27 MHZ, much slower than the main memory chip. Therefore, the secret message displaying dominates the overhead of protection activation. Nonetheless, it is still negligible to the user as compared to the human keystroke speed. The removal of the secret message is not considered as the overhead, because with a high likelihood, it is completed between the user's two keystrokes.

Components	Protection Activation	Protection Deactivation	Keystroke Interception	Displaying Message
Time	$1.71ms$	$3.5\mu s$	$0.12\mu s$	$1.67ms$

Table 1. The performance overhead for password input protection in KGuard.

6.2 Overhead for Password Submission

In the password submission procedure, we evaluate the extra operations introduced by our scheme, i.e. those not appearing in normal web authentication. The extra operations include the extension's HTTP Request event interception and extracting data from the login (POST) request, which cost about $4ms$ in total. Note that the extension is written in JavaScript, whose best timing granularity is in milliseconds. The extra operations also include transferring data between the guest and the hypervisor; HMAC verification for the certificate's trustworthiness. The measurement results are listed in Table 2.

	Event interception and data extraction	Data transferring cost during in hypercalls	HMAC computation
Time	$4ms$	$1.38ms$	$0.02ms$

Table 2. The performance overhead of each component for password submission.

We have also measured the turnaround time to evaluate the overall delay a user may experience with KGuard. The turnaround time refers to the period from the moment when the login button is clicked, to the moment when the browser begins refreshing the page. We have tested KGuard with Twitter and a local web server which resides in the same platform with the browser so that no network delay variation disturbs the results. The results are shown in Table 3. Note that the results from the tests with Twitter are not sufficiently accurate due to the large variance of network round trip time.

	Login without KGuard	Login with KGuard	Extra Cost
Twitter	$1.10s$	$1.11s$	$10ms$
Local Web Site	$201ms$	$207ms$	$6ms$

Table 3. The overall performance measurement in the login procedure.

7 Discussions

7.1 Hypervisor Security

The hypervisor security is the bedrock of the proposed password protection system. It is known that both the code size and the interfaces affect the hypervisor security. According to [2, 20], the size of the source code is proportional to the number of vulnerabilities (bugs). We choose Xen for our prototype building instead of the other mainstream hypervisor VMware ESXi, because the former has a smaller code size according to [26] and is open source. In principle, KGuard can also be built on those tiny hypervisors developed by researchers, such as SecVisor [24], BitVisor [25] and Nova [26]. Unfortunately, they are not supported by the Intel processor used in our platform. As mentioned in [18], interfaces are the main source of critical errors. In the current Xen hypervisor, all default hypercalls for a HVM domain are only used during HVM loading. Therefore, we turn off all of them to enhance security to minimize the attack surface.

In the future work, we aim to reduce the hypervisor code size by removing unnecessary code. Besides the basic hardware virtualization functions, our initial study shows that the functionalities required by KGuard include: 1) memory management, including data transferring and address translation between the guest and the hypervisor; 2) access control on all I/O ports and memory regions; 3) interceptions on interrupts and exceptions; 4) basic crypto algorithms, such as RSA, AES and SHA1; 5) certain instruction emulations; and 6) asynchronization support (e.g., timer).

7.2 Trusted Certificate Updates

The user may need to insert or delete entries in the trusted certificate repository. It is relatively straightforward to add a new trusted certificate. The user simply calculates the HMAC value on a clean system and adds the certificate and its HMAC into the repository.

However, it is costly to revoke a trusted certificate from the repository. One solution is that the user chooses a new HMAC key and re-computes the HMAC tags for all trusted certificates excluding those revoked ones. Once the new key is updated to the hypervisor, the revoked certificates will not pass the verification. Alternatively, the user can prepare a Certificate Revocation List (CRL) whose integrity is protected by the HMAC tag. Whenever the plugin sends the server certificate to the hypervisor, the CRL is attached. The hypervisor then checks whether the certificate in use is on the CRL. Both methods have pros and cons. The former requires more user involvement while the latter increases the hypervisor's code size and causes more runtime overhead.

7.3 Sensitive Keyboard Input Protection

The KGuard system proposed in this paper focuses on password protection. We can easily extend it to protect other sensitive inputs from the keyboard, such as CAPTCHA, credit card numbers or driver license numbers. KGuard is able to intercept and replace the sensitive inputs whenever the user activates the protection. By inserting them back into an SSL/TLS connection or forwarding them to a trusted domain, all sensitive inputs are free from malware attacks.

The challenge is to maintain the user's experience. For a normal password input, the browser only displays a string of '$*$'. The user feels the same even if KGuard replaces the original password with dummy ones. However, for other types of inputs, the user may feel discomfort when seeing dummy characters instead of the expected ones. Another issue on the user interface is how a user determines the correctness of the input, since a wrong key may have been pressed accidentally. One possible solution is that KGuard echoes each input on the screen in the same ways as in the visual verification. Alternatively, KGuard can display the entire input string and ask for user confirmation. This method does not work well for protecting a large amount of sensitive inputs (e.g., private document editing) due to the heavy load on the hypervisor and the slow responses. In addition, it would add too much code into the hypervisor and possibly weakens the security strength.

8 Conclusion

To conclude, this paper has presented a virtualization based password input protection system, which is composed of a novel user-hypervisor interaction channel, a keyboard stroke interception mechanism, and a hypervisor-based SSL client. Our method does not require specialized hardware and is fully transparent to the operating system and the browser. The prototype implementation and testing have demonstrated that the protection system incurs insignificant overhead on the platform and maintains the user-friendliness of password authentication in web services.

Acknowledgements

The authors are grateful to anonymous reviewers for their valuable feedback. This work is partially supported by Centre for Strategic Infocomm Technology (CSIT) Technology Innovation Fund (TIF) Project #PO2011240001 and by Singapore Management University (SMU) Office of Research under the project #12-C220-SMU-003.

References

1. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.
2. Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation0. *Commun. ACM*, 27:42–52, January 1984.
3. Sven Bugiel, Alexandra Dmitrienko, Kari Kostiainen, Ahmad-Reza Sadeghi, and Marcel Winandy. Truwalletm: Secure web authentication on mobile platforms. In *INTRUST'10*, pages 219–236, 2010.
4. Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, March 2008.
5. Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Driverguard: a fine-grained protection on i/o flows. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pages 227–244, Berlin, Heidelberg, 2011. Springer-Verlag.
6. R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proceedings of IEEE Symposium on Security and Privacy*, 2006.
7. CVE-2008-0923. http://cve.mitre.org/cgi-bin/cvename.cgi-?name=cve-2008-0923, 2008.
8. Sebastian Gajek, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. Truwallet: trustworthy and migratable wallet-based web authentication. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC '09, pages 19–28, New York, NY, USA, 2009. ACM.
9. C. Grier, S. Tang, and S. King. Secure web browsing with the OP web browser. In *Proceedings of IEEE Symposium on Security and Privacy*, 2008.
10. IBM. IBM VGA Technical Reference Manual. Website. http://www.mca-mafia.de/pdf/ibm_vgaxga_trm2.pdf.
11. Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.
12. Sawtooth Consulting Limited. CyaSSL Embedded SSL Library. http://www.yassl.com/yaSSL/Products-cyassl.html.
13. Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.
14. Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys'08*, 2008.

15. Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Bump in the ether: a framework for securing sensitive user input. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 17–17, Berkeley, CA, USA, 2006. USENIX Association.
16. Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, February 2009.
17. Microsoft. About the Windows Driver Kit (WDK). Website. http://goo.gl/DfSRi.
18. Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM.
19. A. Oprea, D. Balfanz, G. Durfee, and D.K. Smetters. Securing a remote terminal application with a mobile trusted device. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 438–447. IEEE, 2004.
20. Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM.
21. Wojtczuk Rafal, Rutkowska Joanna, and Tereshkin Alexander. Xen 0wning trilogy. Website, 2008. http://invisible-thingslab.com/itl/Resources.html.
22. Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
23. Limited Sawtooth, Consulting. Ctaocrypt embedded cryptography library. http://www.yassh.com/yaSSL/Docs_CTao Crypt_Usage_Reference.html.
24. Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.
25. Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.
26. Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.
27. The Blue Pill. http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.
28. Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 85, February 2005.
29. Min Wu, Robert C. Miller, and Greg Little. Web wallet: Preventing phishing attacks by revealing user intentions. In *In Proceedings of the Symposium on Usable Privacy and Security (SOUPS*, pages 102–113. ACM Press, 2006.
30. Matei Zaharia, Sachin Katti, Chris Grier, Vern Paxson, Scott Shenker, Ion Stoica, and Dawn Song. Hypervisors as a foothold for personal computer security: An agenda for the research community. Technical report, Jan 2012.