Practical Uses of Virtual Machines for Protection of Sensitive User Data

Peter C.S. Kwan[1] and Glenn Durfee[2]

[1] Electrical Engineering Department, Princeton University **
pkwan@princeton.edu

[2] Palo Alto Research Center
gdurfee@parc.com

Abstract. Systems running commodity software are easily compromised with malware, which may be used by attackers to extract personal information of the users of the systems. This paper presents Vault – a system that uses a trusted software component to prevent the exposure and abuse of sensitive user data in the presence of malware. Users input and store their sensitive data only in the trusted component, which is separated from the commodity system by a virtual machine monitor. We define a protocol framework for the interactions required between different system components in order to protect user secrets, even if the user is running a commodity operating system with arbitrary (and possibly malicious) software load, while introducing minimal changes to the user experience. Our design takes advantage of the isolation guarantees and safe I/O multiplexing of virtual machine technology to attain a high degree of security under a severe threat model.

We demonstrate that our approach is practical by implementing prototypes for two applications: (1) submission of long-term secrets, such as password and credit card data, to a web server, and (2) SSH user authentication using `ssh-agent`. In both cases we made minimal changes to existing software components.

1 Introduction

The widespread use of personal computers running vulnerable commodity operating systems (OSes) has put the personal data of millions of users at risk – data that is easily exploited for identity theft or other fraudulent activities [17]. Attacks that harvest sensitive data[3] from users' computers take advantage of two crucial weaknesses in modern commodity OSes: First, it is notoriously easy to introduce malicious software into a commodity OS through viruses, worms, Trojan horses, and spyware. Second, once running locally, malicious software can easily obtain sensitive information through the use of powerful APIs exposed by the OS, such as keystroke interception and disk I/O. Many security practices, such as the use of secure network protocols and security tokens, become much less effective when the attackers can simply sniff at every key the users type

** Much of this work was done while the author was an intern at Palo Alto Research Center.

[3] In this paper, we use the phrases "sensitive data", "secrets", "sensitive information" and "personal data" interchangeably to refer to a broad class of data users would like to keep private (such as passwords, credit card numbers, and cryptographic keys).

for their passwords, PIN, and credit card numbers, or when the attackers can read any file on the file system. While this is well-known, the superior functionalities and price advantages of modern commodity systems mean they will continue to be in widespread use despite their vulnerabilities.

To address these concerns, we introduce Vault, a virtual-machine-based security system designed to protect sensitive data on commodity systems. Vault uses a virtual machine monitor (VMM) to compartmentalize a physical machine into two virtual machines (VMs). Sensitive data are stored and handled only in the *trusted VM*, while all other computing activities occur in the *untrusted VM*. Users are free to configure the untrusted VM with a commodity OS and a software load of their own choosing. On the other hand, the trusted VM runs a minimal OS with a restricted set of functionalities. To give an idea of what the user experience is using Vault, consider an online shopping scenario. First, a user starts an online shopping session with a web merchant, using their favorite web browser in the untrusted VM. During checkout, instead of entering her credit card number into the browser, she *explicitly* switches to the trusted VM, and inputs it there, where it is then securely transmitted to the merchant's server. Afterward, the system automatically switches back to the untrusted VM to continue the checkout process.

This design is an example of a broader class of systems that protect sensitive data using of small, isolated, trusted components. The trusted components in Vault are the VMM and the trusted VM. Crucially, the trusted VM has a trusted I/O path to the user, especially for receiving confirmations for actions involving the use of sensitive data. This is because the VMM controls the multiplexing of I/O devices, and thus is able to separate the user interactions with the trusted VM from those with the untrusted VM. As our main contribution, we define a protocol framework for the delegation of the handling of sensitive data to the trusted VM. This protocol framework prevents attacks from untrusted components and allows users to guard the use of their sensitive data.

We further show that this framework is practical and can be readily integrated with existing applications. We built a prototype for two of the most common online applications involving user secrets: Web-based online shopping and the `ssh-agent` authentication module used in SSH logins.

The rest of the paper is organized as follows. Section 2 surveys past work in protection of user data in untrusted environments. It is followed by our assumptions on threat and trust in Section 3. Section 4 describes the design of Vault and how different components in the systems interacts to achieve secure use of sensitive data. Then we describe our prototypes in Section 5. In Section 6, we discuss the requirements for widespread adoption of our solution, and argue that it is realistic and achievable. We conclude in Section 7.

2 Related Work

Using separation to improve security of complex systems has had a long history. Small, trusted and tamper-resistant hardware components are used to store sensitive data and handle their operations. Smartcards and secure co-processors [23, 27] are examples of small trusted components designed to be deployed in untrusted hosts. In those designs,

no sensitive data are stored in the host. Instead, the host requests the use of sensitive data, typically cryptographic keys, via an API exposed by the component. However, there is currently no viable way for the trusted component to determine the legitimacy of requests coming from the host, if the component does not have prior state about what can be trusted. For example, the authentication services provided by a smartcard are guarded by a PIN. But since the PIN must be entered by the user *via* the host, it can easily be intercepted by a keystroke-logging malware. Although still unable to access the private key stored in the smartcard, the malware can now make requests, using the sniffed PIN, to the smartcard for operations involving the private key. We have addressed this problem by providing a trusted I/O path between the user and the trusted VM. The trusted VM can then obtain user confirmations for operations involving sensitive data. The work in [10] also employs user confirmation to prevent misuse of sensitive data. However, being a hardware solution, they can only make use of primitive LED and push buttons for user interactions. Our VMM-based approach provides a much more viable user interface for the trusted component.

Isolation mechanisms integrated in the processors have also been proposed. Lee et al. [14], Suh et al. [24], and Lie et al. [15] use cryptographic methods to create isolated and tamper-evident execution environments. Intel [12] and AMD have also proposed curtained memory for the creation of a protected compartment, possibly for a secure kernel enforcing security policies. We focus in this paper not on the mechanisms for isolation, but on how to make use of the isolation guarantees. In this spirit, Jiang et al explore the use of a co-processor to build trust into the services provided by remote servers [13]. Marchesini et al propose the use of the attestation functionality of the TPM [26] chip to attest to the authenticity of a security "Enforcer", which in turn attests to the configuration of the software platform [16]. These work protect users against malicious server operators. We instead protect user's data from malware running on their own computers.

Another approach to provide separation is virtualization. Terra [7] and NetTop [18, 19] use VMM to separate compartments of differing levels of trustworthiness. For example, one VM may be used only for trusted applications handling top secret documents, while another VM, considered less trustworthy, may be used for web browsing. Successful compromise of the web browsing VM does not affect the security of the other VM. The use of VMM for isolation is similar to our work. However, the aforementioned work fails to protect sensitive data that cannot be restricted to a trustworthy VM. For instance, online shopping typically requires entering passwords and credit card information into a web browser loaded with third-party plug-ins. Because of their notorious vulnerability, such browsers usually operate only in the least trusted VMs. Thus, despite the availability of more trusted VMs, users are nevertheless required to input sensitive data into a VM that is much more likely to be compromised.

Proxos [25] allows application designers to specify a subset of the system calls of a commodity OS that their applications do not trusted, and delegates those calls to a trusted private OS, which is separated from the commodity OS by a VMM.

The `factotum` component in the Plan 9 OS represents the closest concept to our work [4]. `Factotum` handles all authentication requests on behalf of the user, optionally requiring user confirmations. Applications wishing to take advantage of `factotum`

needs to be redesigned to delegate the authentication process to `factotum`. However, `factotum`'s trust model includes the Plan 9 OS. Vault addresses a larger set of threats by considering a commodity system in the untrusted VM that may be fully compromised.

3 Security Model

We consider the threats posed by remote attackers on user's long-term secrets used for authentication and e-commerce purposes. We assume attackers can launch attacks from over the network. By exploiting bugs, attackers can compromise a system to install malicious code, such as a keystroke logger. Attackers may also introduce malicious code by employing social engineering techniques. Once installed, we assume that such malicious code may run at the same processor privilege level as the OS kernel, giving it unrestricted access to the private data of all applications, as well as the ability to arbitrarily manipulate application execution. Malware gaining access at this level is not uncommon because most users run with administrator privileges, allowing the installation of kernel components including device drivers. In addition, we also assume that the attackers can intercept and modify all network traffic.

However, we do not consider physical attacks on the hardware platform, such as probing of system buses and keyboard. Also, we do not directly address phishing attacks. Methods to protect against phishing [5, 21] are orthogonal and can be combined with this work.

Under this threat model, we trust the hardware platform. We also trust the VMM and the software that runs in the trusted VM. The remote servers (e.g. Amazon.com), to which the user intends to communicate with and prove possession of his long-term secrets, are also trusted. This means that genuine servers are trusted to handle the secrets correctly. However, the identities of such servers upon connections remain untrusted until proven using certification-authority-based mechanisms provided in secure communications protocols, such as Transport Layer Security (TLS) [6].

4 Design of the Vault System

Figure 1 illustrates the main entities and software components in our design. Vault makes use of a type 1 VMM to provide virtualized hardware. A type 1 VMM runs directly on the hardware [9], whereas a type 2 VMM, such as VMware Workstation, runs atop a host OS. The correct functioning of a type 1 VMM does not depend on an underlying host OS.

The VMM supports two VMs. The untrusted VM runs a commodity OS and an arbitrary application load, chosen by the user. The user conducts most of her computing activities using this VM. The freedom to configure the untrusted VM allows the user to continue to enjoy the superior functionality of a commodity system. Such configuration, however, may contains vulnerabilities. We therefore emphasize that it is completely untrusted, i.e., any malicious software may be operating in the untrusted VM. In contrast, the trusted VM runs a minimally configured OS and only the Vault application. The

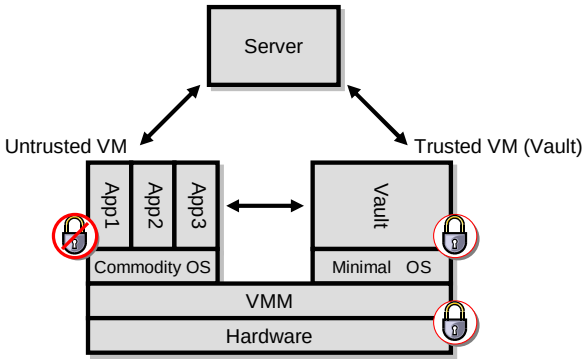

Fig. 1. The software components in Vault. The hardware, VMM, and the software stack of the trusted VM are trusted (indicated by the lock).

Vault application handles long-term secrets of the user. Its provides a graphical user interface (GUI) for user to input the secrets, or to authorize the use of secrets previously stored.

With this organization, the user's secrets are prevented from observation and tampering by any malicious software in the untrusted VM. This guarantee, however, depends on the correctness of the VMM and the software in the trusted VM – they must be correctly designed and implemented in such a way they do not leak any information about the user's secrets. In other words, the VMM, the minimal OS, and the Vault application forms a trusted code base (TCB) of Vault. While it is extremely difficult to fully verify that a piece of software is free of vulnerabilities using today's software engineering methodology, we describe in Section 5 how we take steps to approach this requirement for the TCB.

As mentioned earlier in Section 2 and in [10], the trusted VM needs to be able to differentiate between legitimate and malicious requests from the untrusted VM, in order to prevent misuse of user's sensitive data. Although this is undecidable in general [2, 3], we can make forward progress by making the definition of legitimacy more precise. We say that a request to use or input sensitive data is legitimate when *it is explicitly approved by the user after he or she has been presented with all relevant information associated with the request.*

We recognize that not all users can make good security decisions, even when presented with relevant information. Nevertheless, we view the problem of designing a user interaction model that encourages correct user decisions as a complementary usability problem, a solution to which would work in concert with the software architecture outlined in this paper. The focus of this work is on how to ensure the genuineness of the information presented.

4.1 Trusted I/O and Transition to Vault

In presenting the information and receiving input from the user, we take advantage of the VMM's ultimate control over the I/O devices to present a trusted GUI to the user. There are two aspects to the realization of a trusted I/O path. First, it is enabled by the

trusted multiplexing of the keyboard, video, and mouse by the VMM. Second, and more importantly, we must ensure that malicious software cannot easily spoof a Vault-look-alike in the untrusted VM, enticing sensitive data from the user. In order words, the user must be able to establish which VM she is interacting with.

This can be accomplished by associating special user actions with the transition from the untrusted VM to the Vault. One option to do so is the use of attention key sequences, such as `Ctrl-Alt-Del` required on the Microsoft Windows™ login screen. In Windows, users are trained to associate the attention key sequence with the display of the password prompt. If malicious software spoofs the prompt, the user realizes this by the absence of her special action. Similar to the Windows OS, the VMM can intercept a pre-defined sequence to trigger the transition to the trusted VM, without passing the sequence to the untrusted VM. A more intuitive interface might be a dedicated key for switching to Vault, similar to the password key in [21]. Alternatively, a graphical VM switch can be displayed at the top the screen, as is employed in NetTop [19]. This region must be controlled by the VMM and cannot be obstructed or spoofed by the VMs.

4.2 Protocol Framework for Delegation

To use Vault, the user first initiates a session with a remote server in the untrusted VM. When a long-term secret is requested by the server, she switches to the Vault application in the trusted VM. The Vault application takes over the session from the untrusted VM and requests the secret from the user via a trusted I/O path. The user inputs the secret, which is then transmitted via a secure connection from the Vault application to the server. The system then switches back to the untrusted VM, to continue the session.

In this framework, to ensure that the user is not enticed to submit secrets to malicious servers, the Vault application must (1) verify the information received from the untrusted VM; (2) present relevant information (such as server name) for user confirmation; and (3) establish a secure tunnel to the remote server for the transmission of secrets.

The protocol described below achieves these requirements. To make use of the Vault, existing applications need to be modified to delegate the handling of long-term secrets to the Vault. Figure 2 illustrates the exchanges taken in the protocol.

1. The user begins a session of interaction with the remote server via an application, e.g. a web browser, running in the untrusted VM. She proceeds to a point where long-term secrets is requested. This requests can be a password, a credit card number, or a cryptographic response to a challenge.
2. The user explicitly initiates the transition from the untrusted VM to the trusted VM by pressing a special attention key sequence.
3. In the trusted VM, the Vault application detects that it has been activated. It requests identifiers for the session and the server (e.g. an IP address or a URL) from the application running in the untrusted VM. We refer to these identifiers as `sessionID` and `servername`. Both are sent from the untrusted VM to the Vault application.
4. Using `servername`, the Vault application establishes a secure tunnel with the server. This can be set up using various secure communication protocols, such as TLS [6]. In the secure tunnel, Vault sends the server `sessionID`. Note that this secure tunnel may optionally be relayed via the untrusted VM.

5. The server verifies the `sessionID`. Only if it is valid, the server sends the Vault application information pertaining to that session, as well as a request for the long-term secret. This request may be formatted as an XML form according to some extensible protocols between Server and Vault. Verification by the server prevents the untrusted VM from supplying a malicious `sessionID` at Step 3 above.
6. The Vault application displays information about the server, derived from the secure tunnel. For example, in TLS, the information is the name of the server embedded in its digital certificate. This allows the user to confirm that she is interacting with the intended server. Together with Step 5, this completes the verification of the `sessionID` and `servername` received from the untrusted VM.
7. Once the information is confirmed to be correct, the user responds to the request either by entering the requested secret, or authorizing the use of a secret stored previously by the Vault application. Because of the trusted I/O path, no malicious software in the untrusted VM can eavesdrop or tamper with the user's interaction with the Vault application.
8. The Vault application sends the user's responses to the server.
9. The server concludes by sending the Vault an instruction intended for the application in the untrusted VM. For example, this can be the next URL to be loaded in a browser.
10. The Vault application relays this instruction to the untrusted VM, and signals the VMM to transition back to the untrusted VM.[4]
11. The application in the untrusted VM executes the instruction issued by the server. It continues the session, which is now in a new state after the successful input of the long-term secrets. The user continues to use the untrusted VM for the rest of this session.

With this protocol framework, together with a trusted I/O path between the Vault application and the user, we are able to present genuine information about the session to the user and allow her to approve the use of her secrets, preventing misuses. In addition, the change to user experience is minimal. The only extra step for the user is the transition to the Vault application at Step 2.

5 Implementation

We prototyped Vault along with changes to two popular applications that use long-term secrets. One application is the submission of passwords and credit card data for online commerce with web merchants. The second is public-key user authentication in SSH. We found that only minimal modifications are required to adapt these applications to use Vault. The next section discusses the underlying infrastructure we built to support the Vault prototype.

[4] The reason we relay an instruction from the server to the untrusted VM via the Vault, as opposed to sending it directly from the server to the untrusted VM, is that in stateless applications such as web browsing, a connection between the server and the untrusted VM may no longer exist.

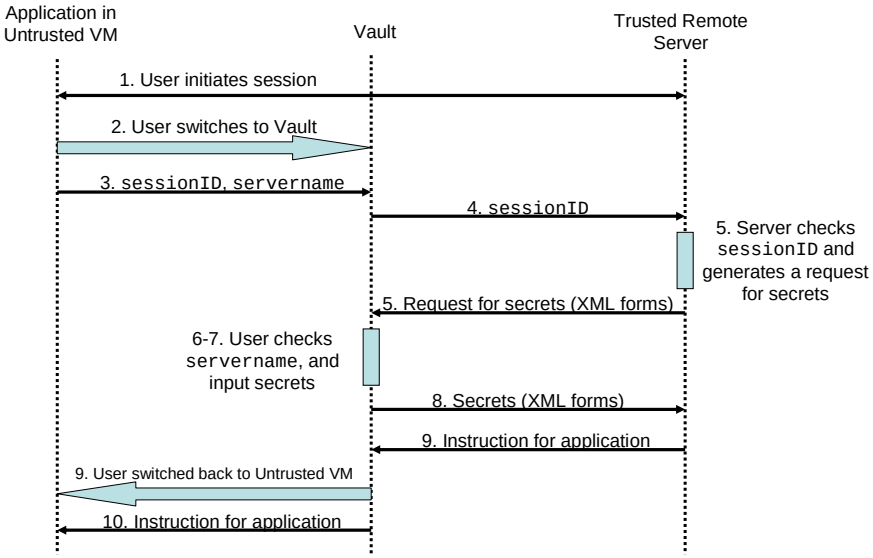

Fig. 2. The protocol framework for delegating the handling of long-term secrets to the Vault application.

5.1 The Virtual Machine Infrastructure

The Xen Virtual Machine Monitor Our design is VMM-agnostic. In the prototype, we use Xen, an open-source type 1 VMM [1] that has improving support for true virtualization using Intel virtualization technology [11]. In Xen's organization, Domain-0 is a trusted management VM, and is started by the Xen VMM at boot time. This VM is trusted to control the rest of the system, such as starting and stopping other VMs. In addition, only Domain-0 has access to all devices on the system. We therefore consider Domain-0, and the services provided by it, a trusted extension of the VMM. In other VMMs, such as VMware ESX Server, the devices are typically controlled directly at the VMM layer.

Operating Systems and Applications in the VMs Both the untrusted VM and trusted VM run Linux in the prototypes. The installation in the untrusted VM is a full client workstation, whereas the trusted VM contains a minimally configured Linux installation with limited network connectivity. Although even our minimial Linux installation probably still contains vulnerabilities, we use this for our research prototype to approximate the requirement of a TCB. (In a production-quality version of Vault, we imagine that a carefully designed and vetted TCB would be used instead. Since very few OS services are required, this could be a very small TCB.) In our prototype, only the Vault application runs in the trusted VM, and network connectivity is limited to only what is needed by the Vault application. The Vault application is minimally designed to provide only functionalities required to handle user's long-term secrets. By limiting functionality and connectivity, we argue that even our research prototype is sufficient to neutralize a large class of potential attacks on the trusted VM.

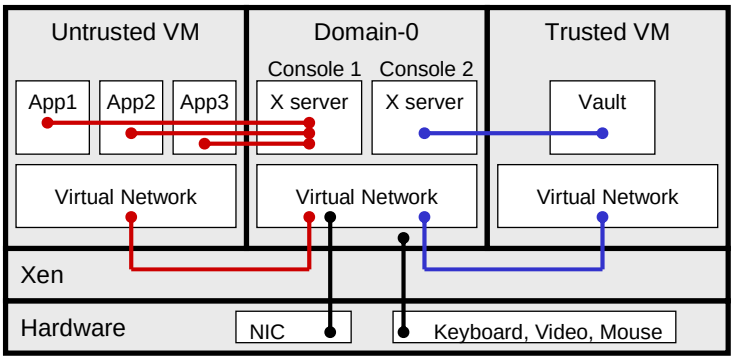

Fig. 3. The setup of the virtual machines. Applications in each VM display in a VM-specific virtual console in Domain-0. Each VM also have an isolated virtual network connection to Domain-0.

Multiplexing of Trusted I/O One of the most important requirement for the Vault is the trusted I/O path to the user. We leverage on the fact that the trusted Xen Domain-0 controls Linux's virtual consoles (accessible by the `Ctrl-Alt-Fn` sequence) to provide an attention key sequence for switching between the VMs. We run X servers on virtual console 1 and 2 of Domain-0. These X servers receive commands from applications in the VMs via the X Display Manager Control Protocol (XDMCP) and the X11 protocol [29]. Figure 3 illustrates this organization. Because the `Ctrl-Alt-Fn` sequences are serviced by Domain-0 without passing to the VMs, the transition between VMs is non-bypassable. Note again that X11 has not been verified to be bug-free, and we use it for the purpose of illustrating the configuration of the trusted I/O path. Utlimately, security depends on the safe I/O multiplexing feature provided by the underlying VMM. While Xen does not provide the highest assurance, commerically available VMMs do, as illustrated by their uses in NSA's NetTop architecture [19].

Virtual Network Configuration The communications between the trusted VM and untrusted VM are handled by a virtual network exposed by the Xen VMM. The virtual network topology must ensure that the untrusted VM cannot eavesdrop on the traffic from the trusted VM. Therefore, the two VMs must never be connected to the same bridge. In our implementation, the trusted Domain-0 runs as a router and a network address translator (NAT). Two virtual interfaces are defined in Domain-0 for isolated connections to the two VMs (Figure 3).

5.2 Prototype 1: Submission of long-term secrets to a web merchant

We consider the everyday scenario of entering passwords and credit card numbers for online shopping. The system we implemented is generic enough for any e-commerce web site to take advantage of Vault. We first describe the new user experience:

User Experience The user starts by conducting normal online shopping activities in the untrusted VM. Once the user navigates to a page requesting a long-term secret, such as a credit card number at the check-out page, she presses an attention sequence to switch

into the trusted VM. In our implementation, the attention sequence is `Ctrl-Alt-F10`. This switches the display to virtual console 2 – that of the trusted VM. In the trusted VM, the Vault application displays the name of the web merchant and its requests for the secret. The user checks the name of the merchant. If it is correct, she inputs the secret in the trusted VM and authorize it to be sent to the merchant. Next, she is automatically switched back to virtual console 1 of the untrusted VM. The browser loads a new page, which may indicate that the credit card information has been received. The user continues with the rest of the session in the untrusted VM, making full use of the rich functionality provided by the commodity OS in the untrusted VM.

As the user go through these steps, she participates in the protocol described in Section 4.2. Note that the only additional step required of user is the explicit switch to the trusted VM before she enters her credit card number. Next, we describe the modifications required in the untrusted VM and the server.

Communications between the Trusted and Untrusted VMs As part of the protocol, the Vault application needs to get information from the web browser running in the untrusted VM. We augmented the Firefox browser with a browser extension that listens for connections from the Vault. Upon a connection from the Vault application, the browser extension sends it `sessionID` and `servername` (step 3). The values of `sessionID` and `servername` are specified by the web merchant as hidden input fields in an HTML form, for example:

```
<input type=hidden name=vault_sessionID value=165996028513308>
<input type=hidden name=vault_servername
       value='https://www.amazon.com/vault'>
```

Thus, the browser extension simply reads these values from the HTML page and sends them to the Vault application. Note that these values may be tampered with in the untrusted VM. Therefore, they are verified by the server at step 5, and by the user at step6 of the protocol in Section 4.2. After sending these to the Vault application, the browser extension waits for an instruction that the Vault received from the merchant (Step 9). In this prototype, the instruction is simply a URL link to a web page acknowledging the receipt of the credit card number. The browser extension completes the protocol by loading this page in the browser.

Server Modifications On the server side, hidden input fields for `sessionID` and `servername` are inserted in the check-out web page. The secure communications between the Vault application and the merchant is handled by a new set of handlers (steps 4, 5 and 8). We envision the Vault-server communications be based on a standard in XML format, possibly established by industry consortia. In addition, the server updates an internal session database, in order to keep track of the interactions with the Vault application and the session originally started by the user in the untrusted VM. Such session tracking feature is commonly employed in actual e-commerce sites. It is therefore straightforward to augment it with data for Vault.

The Vault Application When user switches to the trusted VM, the Vault application detects it and establishes a connection to the Firefox extension in the untrusted VM to retrieve `sessionID` and `servername`. Using `servername`, it establishes a separate HTTPS connection to the web merchant. The Vault rejects the connection if the

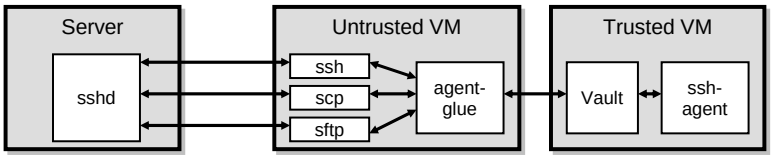

Fig. 4. The glue component in the prototype for SSH user authentication.

certificate is not signed by a known certificate authority. In this connection, the Vault sends the `sessionID`, and receives an XML forms requesting for long-term secrets. Next, the Vault prompts the user for the requested long-term secrets. Importantly, it also displays the name of the merchant in its GUI. This is the Common Name in the certificate presented by the merchant for the HTTPS connection. This allows the user to confirm that she is communicating with the right merchant. After user's input, the Vault sends the long-term secrets to the merchant via the HTTPS channel. As a last step, it receives a new URL from the merchant. This URL is the next page to be loaded in the browser running in untrusted VM. Finally, the Vault application triggers a switch back to the untrusted VM by making a request to a helper program resident in Domain-0. This helper program writes to `/dev/console` to trigger a virtual console switch back to the untrusted VM.

5.3 Prototype 2: Public-key User Authentication in SSH

Overview of SSH Using `ssh-agent` The SSH protocol supports many user authentication options. One of them uses public key cryptography [30]. The authentication private key is stored locally on the client computer, and is typically encrypted using a user-selected passphrase. At *every* SSH login, the `ssh` client prompts the user for the passphrase to unlock the private key, which is then used to generate a response (a signature) to a challenge posed by the remote SSH server. The `ssh-agent` application streamlines this process by enabling a single sign-on feature. At launch, `ssh-agent` prompts user for the passphrase *once*. It then runs as a background process, listening on a Unix socket. Client applications (such as `ssh`, `scp`, etc) engage `ssh-agent` via the socket to authenticate the user to remote SSH servers. Essentially, `ssh-agent` signs, using the unlocked private key, any binary data that is passed to it from any client. This "signature service" is unguarded once the passphrase has been entered at the launch of `ssh-agent`.

In our threat model, there are two attacks on an `ssh-agent` running in an untrusted VM: (1) an attacker who has compromised the OS can read and write the private key files stored locally, and (2) short of tampering with the private keys, a malicious program simply makes use of the unguarded signature service to login to a remote server where the user has an account, impersonating the user.

Moving `ssh-agent` *to the Trusted VM* To defend against the first attack, `ssh-agent` and its public-private key database are relocated to the trusted VM. This step requires no changes to `ssh` and `sshd`. We first relocate `ssh-agent` to the trusted VM, and with it all the private keys of the user. We then implemented a new glue process, `agent-glue`, in the untrusted VM, as shown in Figure 4. `agent-glue` disguises

as `ssh-agent` by opening a Unix socket and setting up the appropriate environment variables. It relays any connections to the trusted VM. In the trusted VM, the Vault application waits for connections from `agent-glue`, and in turn relays the connections to the genuine `ssh-agent` running there. With this arrangement, the private keys and the passphrase are protected from the untrusted VM. However, the *use* of them is still unguarded.

Securing against Misuse of Private Key To defend against the second attack, we need to add a confirmation stage in the Vault application. Each time the user logs in to a remote server using SSH, she needs to explicitly switch to the trusted VM to confirm the use of her long-term secrets, using the protocol framework described in Section 4.2. However, with this protocol, `sshd` needs to be modified significantly in order to support a separate secure tunnel between the Vault application and `sshd`. A simpler approach can be devised by observing that the secure tunnel is a requirement only if the secrets need to be transmitted to the server in their actual forms, such as credit card numbers. For SSH, there is no such requirement because the secret – the private key – is never actually transmitted to the `sshd` server. The user only needs to prove the possession of the private key by furnishing a valid signature. This property is generally true for challenge-response authentication protocols. In Appendix A, we consider a subtle attack on this class of authentication under our threat model, and define a simplified delegation protocol framework that eliminates the Vault-server secure tunnel. The main requirement of the simplified protocol is that the authentication scheme must bind *both* `sessionID` and `servername` in the signature. However, SSH protocol only binds the `sessionID` in the signature. But we found that it is fairly easy to argument the SSH programs to fulfill the requirement without modification to the SSH protocol.

Modifications to `ssh` *and* `sshd` During user authentication, the `ssh` client proves the possession of the private key by signing a pre-defined data structure that includes the `sessionID`, among other data. However, `servername` is absent from the structure. We need to augment this data structure with the `servername`. Conveniently, the `sessionID` field is variable-length. The `ssh` client can thus prefix the original `sessionID` field with `servername` and a delimiting character. This augmented structure is sent to `ssh-agent` in the same manner as the current challenge – as one binary data block. The Vault application intercepts this message to extract `servername` in order to display it for user confirmation, and only forwards this whole message to `ssh-agent` for signing if the user confirms that this is the intended use of her private key.

On the server side `sshd` checks *both* `sessionID` and `servername` to ensure that the signature is the correct authenticator intended for it. We estimate that this change requires only modest modifications to the code of `ssh` and `sshd`.

Limiting the Danger of Session Hijacking Despite providing strong protection for long-term secrets, our design does not protect short-term secrets used in the untrusted VM, most notably the session keys used in `ssh`. This implies that sessions are still vulnerable to *session hijacking*, whereby an attacker takes over a session (possibly without the user's awareness).

Because our threat model implies that no application in the untrusted VM is safe, there is nothing that can be done to prevent session hijacking. But we would like to design our system to minimize the damage caused by session hijacking, and in particular, prevent situations in which hijacking can be used to modify the server's notions of the user's long-term secrets.

The session opened up by `ssh` is a shell running on the server with the user's full privileges, allowing, for example, the modification of the `~/.ssh/authorized_keys` file. This could allow an attacker to insert or remove entries in the list of public keys authorized for user login. Therefore, in order to take advantage of our construction for the full protection of long-term secrets, the session interface must be limited in what the user can operate on long-term secrets. All operations related to the integrity of the long-term secrets must be arranged so that they are carried out via the Vault. This implies that the list of public keys accepted for user login should only be updateable using a protocol that uses the Vault for user confirmation.

6 Discussion

As discussed in the introduction, widespread adoption of our design rests on several working assumptions, which we discuss in this section.

Application Adaptation Application designers must make careful decisions about what data needs to be protected in their systems, and make changes to follow the protocol framework we propose. Although modifications are clearly needed, the advantage of being able to provide a more secured environment to the customers can be a major incentive. In addition, our implementation shows that such modifications are simple. For example, the applications using the SSH protocol discussed in Section 5.3 and refined in Appendix A require only minor changes on the code of `ssh`. For applications related to online commerce (Section 5.2), a bit more work needs to be done to add a mechanism to delegate handling of long-term secrets to the Vault application. Fortunately, the extensible nature of Firefox and other modern web browsers makes it straightforward to augment the browser with a Vault-aware extension. Our prototype extension has less than 60 lines of Javascript code, and our server side support totals only 200 lines of Python code.

Changes in User Behavior Users must be willing to minimally change their behavior. We also do not expect this to be a significant hurdle to deployment, since the adjustments in behavior are small: the user must learn to switch to the Vault application for the entry of sensitive data. With appropriately designed web pages, the user can be explicitly prompted to perform this action, so the user does not need to explicitly commit this new interaction to memory.

Widespread Adoption of Virtualization The Vault system fundamentally relies on a virtual machine monitor running on the hardware. Although we can only conjecture about the future, we feel the prospects of widespread adoption of VMM technology are very good. Hardware support for virtualization is improving [11]. There are other

compelling applications, such as mobility [20], intrusion detection [8], and software maintenance [22, 28] that will help drive the demand for virtualization technology.

7 Conclusion

In this paper, we present a novel design for using virtual machine technology to protect user sensitive information, such as passwords, credit card data, and cryptographic keys. Our approach makes use of the strong isolation guarantees of a virtual machine monitor (VMM) to separate an untrusted, commodity operating system from a trusted "Vault" handling long-term user secrets. We define a protocol framework that can be employed by any application to use the Vault for the safe handling of user long-term secrets.

We achieve several key design goals: we allow the users to continue to use commodity operating systems with arbitrary software configuration; we ensure that software in the untrusted environment cannot observe or tamper with the secrets stored in the Vault; and we ensure that the untrusted domain cannot make use of the Vault's long-term secrets without user confirmation. In addition, the changes to user experience are minimal.

Our implementation shows that this design is practical. We use the Xen virtual machine monitor to provide strong isolation guarantees and a trusted GUI. We also found that adapting existing applications to utilize the Vault requires only small changes.

Acknowledgments

The authors would like to thank Dirk Balfanz, Diana Smetters, and Hao Chi Wong for their valuable comments on earlier drafts of the paper. We would also like to thank Professor Ruby Lee of Princeton University for her encouragement and support.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. 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 Press.

[2] D. M. Chess and S. R. White. An undetectable computer virus. In *Virus Bulletin Conference*, Sept. 2002.

[3] F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, 1987.

[4] R. Cox, E. Grosse, R. Pike, D. L. Presotto, and S. Quinlan. Security in Plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, Berkeley, CA, USA, 2002. USENIX Association.

[5] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88, New York, NY, USA, 2005. ACM Press.

[6] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, Jan. 1999.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2003. ACM Press.

[8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Internet Society's 2003 Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, 2003.

[9] R. P. Goldberg. Architectural principles for virtual computer systems. PhD thesis, Harvard University, 1972.

[10] P. Gutmann. An open-source cryptographic coprocessor. In *Proc. 9th USENIX Security Symposium*, Denver, CO, Aug 2000.

[11] Intel. Intel Virtualization Technology Specification for the IA-32 Intel Architecture, 2005.

[12] Intel. Lagrande technology. http://www.intel.com/technology/security, 2005.

[13] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *17th Annual Computer Security Applications Conference*, Dec. 2001.

[14] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 2–13, June 2005.

[15] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 168–177, 2000.

[16] J. Marchesini, S. W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-source applications of TCPA hardware. In *20th Annual Computer Security Applications Conference*, Dec. 2004.

[17] B. McCarty. Automated identity theft. *IEEE Security & Privacy Magazine*, 1(5), 2003.

[18] R. Meushaw, M. Schneider, D. Simard, and G. Wagner. Device for and method of secure computing using virtual machines. U.S. Patent no. 6,922,774, July 2005.

[19] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. NSA Tech Trend Notes, 9(4), http://www.vmware.com/pdf/TechTrendNotes.pdf, 2000.

[20] M. T. Raghunath, C. Narayanaswami, C. Carter, and R. Caceres. Reincarnating PCs with Portable SoulPads. Technical Report RC 23418, IBM Research, June 2005.

[21] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proc. 14th USENIX Security Symposium*, Aug. 2005.

[22] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[23] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(9):831–860, Apr. 1999.

[24] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 25–36, 2005.

[25] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proc. 7th Symposium on Operating Systems Design and Implementation*, Nov 2006.

[26] Trusted Computing Group. TCG TPM Specification Version 1.2 Revision 85. http://www.trustedcomputinggroup.org, Feb. 2005.

[27] J. D. Tygar and B. Yee. Dyad: a system for using phyiscally secure coprocessors. In *Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment*, Apr. 1993.

[28] VMware. VMware ACE. http://www.vmware.com/products/desktop/ace_features.html.
[29] X.org Foundation. Documentation. http://www.x.org/, 2005.
[30] T. Ylonen and C. Lonvick. SSH authentication protocol. IETF Internet Draft, Mar. 2005.

A Simplified Delegation Protocol Framework

The protocol framework described in Section 4.2 is applicable to all types of long-term secrets. In particular, the Vault-server secure tunnel allows the Vault application to transmit secrets to the server in their actual form. However, if the exchange does not involve the secrets in their actual form, the secure tunnel can be eliminated. This is usually the case in authentication systems that use public-private or shared secret keys. We define a streamlined protocol framework that eliminates the Vault-server connection. We use SSH authentication as an example as we describe the design of the protocol.

In challenge-response authentication schemes such as the SSH authentication protocol [30], the secret – the authentication private key – is never sent to the server in its actual form. Only an authenticator, which usually is a signature of a challenge, is sent to the server.

We begin by exploring a subtle attack: Consider a scenario where the user has accounts with servers A and B. With both servers, she uses the same public-private key pair for authentication. We also consider an attacker who attempts to impersonate the user by logging on one of these servers. With user confirmation in the Vault, the attacker cannot do so readily. However, the attacker can mislead the user into approving a bogus login attempt by carrying out a man-in-the-middle attack. Consider a compromised `ssh` client in the untrusted VM. The user uses it the login to A. Instead of requesting a challenge from A, the malicious `ssh` client requests one from B, and pass it to the Vault for the generation of a response. The attack succeeds because the user is misled to believing that she is approving for a login to A.

To overcome this attack, the user intention must be bound to the response. In this case, the user intends to login to server A. Recall, from Section 4.2, that both `sessionID` and `servername` are supplied by the untrusted VM and so they must be verified by the trusted entities. `sessionID` is verified by the server at Step 5, whilst `servername` is verified by the user using the server certificate of the secure tunnel between Vault and the server, at Step 6.

Since the Vault-Server tunnel is to be eliminated, there is no server certificate. Therefore, `servername` must be verified by some other means. We observed that `servername` can indeed be verified by the trusted `sshd` itself. It can be carried out as follows: (1) the challenge supplied by `ssh` must contain *both* `sessionID` and `servername`; (2) Vault displays `servername` for user confirmation; (3) Vault signs the challenge, binding the user confirmation of `servername` to the signature; (4) finally, the trusted `sshd` verified that the signature is intended for it by verifying the correct `sessionID` and `servername` are both embedded in the signature.

The signature generated at Step 3 above can be sent to `sshd` via the untrusted VM, thus removing the need for a Vault-Server tunnel. The full simplified protocol is described below. Figure 5 illustrates it.

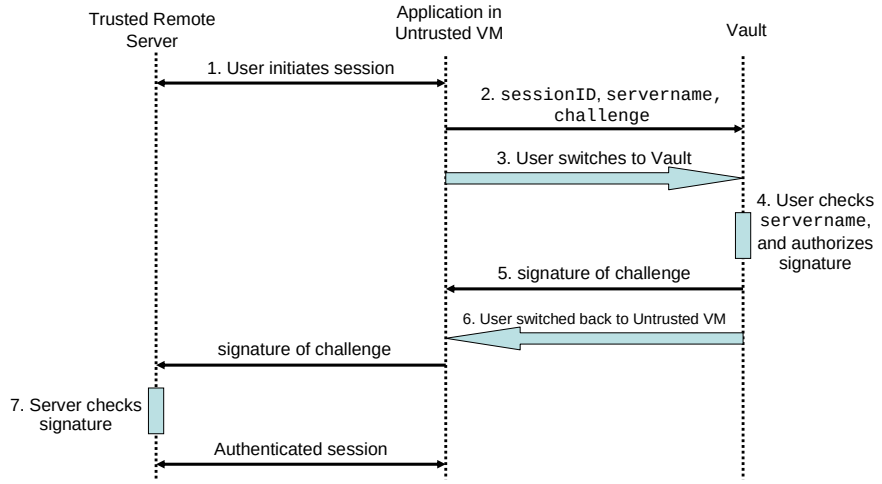

Fig. 5. The simplified protocol framework for challenge-response-based authentication protocols.

1. The user begins a session of interaction with the server via an application running in the untrusted VM. She proceeds to a point where she needs to generate a response to a cryptographic challenge.
2. The application sends the Vault `sessionID`, `servername`, and the challenge.
3. The user explicitly initiates the transition to the trusted VM. In its GUI, the Vault application displays `servername` for user confirmation.
4. Once the information is confirmed to be correct, the user authorizes the generation of a response using some long-term secrets, and incorporates in the response both `sessionID` and `servername`.
5. This response is passed back to the application in the untrusted VM for relaying to the remote server.
6. The Vault application signals the VMM to transition back to the untrusted VM.
7. The server verifies the response, `servername` and `sessionID`. If all are correct, it continues the session.

The above procedure is applicable to authentication protocols using public-private keys or any other shared cryptographic secrets. It is not applicable to long-term secrets that must be transmitted in their actual form, such as credit card numbers. For those the original protocol framework described in Section 4.2 should be used.