HyperPass: Hypervisor Based Password Security

James "Murphy" McCauley, Radhika Mittal

Abstract

Passwords are the linchpin in the security of an increasing number of online services – services that range from social networking to business communication to banking. Given their importance, it is unfortunate that passwords are relatively easily stolen using a number of different types of attack. We introduce HyperPass: an approach and proof-of-concept system that aims to prevent some of these attacks by moving passwords from a user’s normal operating environment into a secure hypervisor. Notably, this is done while maintaining compatibility with existing online services, applications, and operating systems.

1 Introduction

Passwords are the linchpin of online security. Certainly, there are other major technologies involved in cybersecurity. And certainly, stronger types of authentication exist (e.g., hardware tokens, multifactor authentication, and cryptographic methods) [25]. However, in most case it is largely just a string of seven or eight characters [4] that keeps one user from pretending to be another when interacting with an increasing number of online services – services which touch our lives in an increasing number of ways, from our personal lives, to our business practices, to our banking and finances.

Considering their importance, it is unfortunate that passwords are relatively easy to steal. Indeed, attackers have a variety of ways of attaining passwords:

**Attacking servers:** Attackers have been successful in stealing passwords directly from the services for which the passwords are intended to protect access; indeed there have been a number of high-profile cases of this [31].

**Network-based attacks:** In theory, Transport Layer Security provides strong protection from passwords being “sniffed from the wire”. In practice, however, software bugs [15] and compromised Certificate Authorities [29] have shown that it is not airtight. Clever attacks on TLS usage such as “SSL stripping” [18] weaken it further.

**Phishing attacks:** It has been shown that it is quite possible to fool users into divulging passwords and other private data [11]. While some are quick to dismiss such attacks as “user error”, phishing can be coupled with network-based attacks or can incorporate techniques such as homograph domain names to create user experiences that are very difficult to differentiate from legitimate ones.

**Attacking hosts:** By compromising a user’s machine, passwords can be stolen directly in any of several ways, e.g., by examining HTTP post data, reading them from browser password managers, or logging keystrokes. These techniques have been used by botnets (the Torpig botnet alone steals a password every second [30]) as well as by off-the-shelf “spyware” which has even been pre-installed on rental computers [24]. While preventing this sort of host compromise is an ongoing effort by both industry and academia, it continues to be an elusive goal.

Additionally, passwords have shortcomings stemming from the fact that they must be remembered [8]. This constraint leads many users to select “weak” passwords, which automated tools can brute-force in a tractable amount of time [30]. This also leads to many users reusing passwords across services, meaning that an attacker who has stolen a single password for a user may have actually gained access to that user’s accounts on multiple services. Finally, passwords are replayable, but few users change them regularly due to the burden of remembering new ones. Making all of this worse: users may receive little or no indication that their passwords have been stolen until it is too late.

In this paper, we introduce HyperPass – the primary purpose of which is to protect against host-based password theft while using unmodified operating systems, applications, and online services. We do this by leveraging a hypervisor-based approach that isolates passwords from the user’s operating system and applications. In addition, we explore how HyperPass can be used to confront some of the dangers of phishing, network-based attacks, and other sources of password insecurity.
The remainder of this paper is organized as follows: In § 2 we examine some of the related work and its relationship to HyperPass. In § 3, we explain the basic architecture of HyperPass and its different components. This is followed by a security analysis in § 4. We describe our proof-of-concept implementation of HyperPass in § 5 and evaluate its performance overhead in § 6 before concluding in § 7.

2 Related Work

One way to prevent credentials from falling into the wrong hands is to change their nature – moving from simple usernames and passwords (which can be stolen and replayed) to something else. Various schemes have been proposed and deployed in this vein [25], including security tokens, smart cards, and biometric authentication. Often, these alternate approaches are used as part of Two-Factor Authentication [27], which requires some combination of a knowledge factor (such as a password), a possession factor (such as a smart card), and an inherence factor (such as biometric data). Due to their recent widespread availability, cellular telephones are now often used as a possession factor, for example by Google [9] and banking websites. At a high level, our approach is similar to the above – prevent usable credentials from falling into the wrong hands. However, our approach does not by altering the credentials themselves (which takes the cooperation of both the user and the service provider) but by guarding existing credentials from attackers (which does not).

Like HyperPass, other systems attempt to hide passwords from malicious software. For example, password managers in general provide some protection against key loggers, as passwords need not be typed over and over. Some password managers, such as the one in Mozilla Firefox and Apple’s Keychain, store passwords in encrypted form so that a simple memory or disk dump will not expose them. Still others [1] attempt to hide passwords “in plain sight” by saving thousands of decoy passwords along with real ones. Another method – onscreen keyboards [10] [12] – provides further key logger evasion. All such methods are relatively weak evasions: while the passwords may be entered or stored in a way that is resistant to some forms of inspection, eventually the passwords all pass through application code in their unencrypted form, at which point they are vulnerable. HyperPass, by contrast, keeps passwords secret even from application code.

HyperPass has the most in common with other hypervisor-based security approaches. A security model involving two machines – a “red” machine for general use and a “green” machine for accessing sensitive data – was articulated in [7], and this basic idea has since been applied to virtual machines [23]. Overshadow [2] provides a twist: applications see their own resources as usual, but the operating system and other applications see those resources in an encrypted form. CloudTerminal [13] proposes a “secure thin terminal” environment which provides access to remote applications run by services, but isolated from the ordinary operating system. In general, these approaches provide some number of isolated environments, with the idea that an attacker may compromise one, but still be unable to access sensitive information in another environment. This isolation comes at a cost in terms of user experience – a user must adjust the way in which he or she works to account for isolated user interfaces and limitations in how data can be shared between these environments. HyperPass isolates along a different axis: instead of isolating applications from each other, we isolate just passwords from everything else. In particular, this means that existing applications can continue to share data just as they always have – it is just that none of them can access passwords. Additionally, where CloudTerminal requires the cooperation of services, HyperPass does not. Recently, others have suggested storing only passwords in an isolated environment [3]. Conceptually, this is very similar to HyperPass. Unlike HyperPass, however, their method does not operate at the network level, requiring browser plugins or other changes to applications. Furthermore, the authors make no attempt to support existing services or tackle any attacks except for key loggers.

Under the assumption that hypervisors can be made to securely isolate data on a machine (an assumption we share), researchers have suggested a broad agenda for further research in the use of hypervisors to improve the security of personal computers, including a proposal for authentication [32]. Indeed, our work is largely inspired by this agenda, though we stray from the “Hypervisor-Assisted Authentication” method it proposes, which requires cooperation of the operating system, applications, and services, and instead focus on a more incremental approach which does not require cooperation from any of these.

3 HyperPass Design

We take cues from recent work [13, 32] which explores using a secure hypervisor to provide security improvements to existing operating systems as discussed in § 2. Specifically, HyperPass necessitates two environments (Figure 1). The first of these is a User environment which runs all of a user’s existing software – their operating system, applications, and (potentially) password-stealing malware. The second is the HyperPass environment which stores passwords (and the HyperPass code itself).
Our focus is on securing passwords with services that use Transport Layer Security, as passwords that are sent in the clear are quite susceptible to theft in network attacks which are outside of our ability to fix. Thus, HyperPass must perform a benign man-in-the-middle attack on TLS connections as shown in Figure 2.

To achieve this, HyperPass essentially becomes its own Certificate Authority with its own root signing key, which must be accepted by the operating system and/or applications in the User environment.\footnote{Indeed, this is the single “change” we make to the User environment.} To intercept a TLS connection, HyperPass: (1) makes its own TLS connection to the service at the appropriate destination IP address and port, (2) validates the service’s certificate, (3) reads the service’s certificate and creates a duplicate (most significantly, it copies the Common Name), (4) signs this new certificate with the HyperPass root certificate, (5) accepts the TLS connection from the User environment using the new certificate, and (6) passes data back and forth between these two connections, possibly monitoring or altering data.

As briefly mentioned in § 1, issues with both certificate validation and certificates themselves have been shown to be exploitable. Because HyperPass maintains service-specific information (detailed further in § 3.3), it can optionally match the exact certificate known to be used by a specific service in step 2 – not simply one which appears to be valid. This mitigates a number of such attacks, and we further discuss it in § 4.2.

3.2 Password Management / User Interaction

Early on, we conceived of HyperPass largely as a user interface for entering passwords that was isolated from malicious software in the User environment. The user would know their real passwords, but would not enter them directly in the User environment – only in a secure UI provided by HyperPass which would be displayed when an authentication event was identified on the network. We imagined this UI would contain a “reverse password”\cite{13} – an image known only to HyperPass and the user. By recognizing this image, the user could be assured they were, in fact, typing the password into HyperPass and not some application posing as HyperPass in order to extract passwords from the user.

However, other work suggests that such a user interface may not be sufficient in the face of human rule-based decision making. Specifically, Karlof et al.\cite{11} state that “since a wide range of Web sites require a user to log in before she can do something interesting, many users have developed a rule of the form ‘if (login form) then (enter username/password)’ and will aggressively apply it when they encounter login prompts on Web pages which on the surface appear familiar, legitimate, or trustworthy.” This motivated us to pursue an alternative approach, where the user does not know their own passwords. In this approach, when a user first attempts to log in to a new service, HyperPass generates a new random password and automatically changes it on the server without ever informing the user what this password is. When a user attempts to use the associated account, instead of prompting for a password, HyperPass will
simply prompt the user to confirm that it is his or her intention to log in to the relevant account; if the user responds affirmatively, HyperPass automatically sends the correct password. The need for this confirmation UI and various other implications of this approach are further discussed in § 4.4, but note that in this scheme, the only time a user enters a password is when first using an account with HyperPass, after which the password is immediately changed – even if an attacker should intercept it, it immediately becomes irrelevant.

3.3 Service Modules

Different services use different protocols (e.g., HTTP, IMAP, SMTP, proprietary IM protocols), and HyperPass needs to understand how each of these handles authentication so that it can replace the dummy passwords with the real ones. Additionally, while web-based services all use the same protocol, they all do authentication slightly differently – for our purposes, each of these is really a slightly different protocol. To confront this, every service HyperPass supports has a corresponding Service Module which understands the authentication mechanism used by that service. Each Service Module is comprised of the following components: (1) the domain name and TCP port used by this service for authentication, (2) password replacement logic, and (3) logic to generate a new, strong password and instruct the service to change passwords. A Service Module may also include: (4) logic to remove improve the user experience (e.g., removing the meaningless password field from HTML login forms), and (5) a hash of the exact TLS certificate used by the service (further discussed in § 4.2). Figure 3 shows the essential flow of data through a HyperPass Service Module.

4 Security Analysis

We analyze the password security provided by HyperPass by considering three attack models for stealing passwords – those revolving around compromised hosts, network-based attacks and attacks against weak passwords.

4.1 Compromised Host

Protection from password theft in the case of a compromised host is the major motivation behind HyperPass. As passwords are used, they travel through multiple subsystems of the software environment. For example, a common pattern is for a password to be entered via the keyboard, stored on disk by an application or password manager, loaded into application (e.g., browser) memory, and ultimately encrypted by a TLS library. Depending on the exact way in which a host is compromised, a password may be stolen at any of these or other points [26], so we assume a worst case scenario: an attacker who has complete control over the entire User environment: its operating system, applications, and data. In HyperPass, passwords for supported services never even enter the User environment – only in the secure HyperPass environment. Thus, even such a strong attacker would be unable to access passwords unless he or she was able to violate the hypervisor-based isolation.

4.2 Network Attacks

HyperPass also confronts some network-based attacks, specifically phishing and man-in-the-middle attacks. We make the assumption that authentication is intended to be done over TLS; if authentication is done “in the clear”, there is little that can be done to prevent theft in the face of an attacker with appropriate position in or control over the network. We also assume that although the attacker can drop, create or alter any traffic arbitrarily, he cannot break TLS: once the data has been encrypted using TLS, the attacker cannot decrypt it without the correct key. Finally, we assume that the attacker cannot take advantage of vulnerabilities at the server, as HyperPass cannot fix broken services.2

Under these assumptions, the TLS certificate offered by a service should prevent both phishing and man-in-the-middle attacks, and an attack can be successful only if the user somehow fails to notice that there is a problem with the certificate, e.g., that it is for the wrong domain, that it is issued by an unreliable registrar [29], or that there is no certificate at all due to “SSL Strip” attacks [18]. We do not intend to blame the user here; many of these attacks

---

2A network attacker also cannot take advantage of vulnerabilities at the client – such an attack is simply a weak version of a compromised host attack.
are so subtle as to require unreasonable effort on behalf of the user to notice.

HyperPass prevents such phishing attacks because the user does not know their own password: no matter how convincing a phishing site appears to be, the user simply cannot divulge their password. Moreover, HyperPass will not send passwords to phishing sites because look-alike and homographic domain names which may appear the same to humans do not look alike at the byte level at which HyperPass makes its comparisons.

HyperPass Service Modules have an optional mechanism (mentioned earlier) that can help to protect against man-in-the-middle attacks. As Service Modules contain service-specific information, they can also contain a hash of the exact certificate that a service uses. An attacker may be able to present a certificate that would look legitimate to an application (e.g., by having it issued by a compromised CA, or by taking advantage of bugs in certificate validation [15]), but these would not fool a Service Module looking for a particular certificate known to be associated with the service.

4.3 Weak Passwords

Weak passwords that succumb under guessing or brute force are another cause of successful attacks against online services. Password analysis done in [30] shows that over 40% passwords are crackable within 75 minutes and over 28% people reuse their passwords across sites. HyperPass generates strong, unique passwords for every service, protecting against such attacks.

4.4 Physical Possession Attacks

As previously mentioned, authentication mechanisms are generally considered to be based on knowledge, possession, or inherence. Passwords fall into the first category, but as described in § 3.2, with respect to the user, HyperPass eliminates these because there are indications that it is simply too easy to convince users to share this knowledge. Instead, only HyperPass (and the service provider itself) has “knowledge” of the password. In fact, the goal of this aspect of HyperPass is to exchange the knowledge factor of passwords for a possession factor: physical possession of the actual computer running HyperPass. Here we examine two outcomes of this exchange.

4.4.1 Zombie Logins

HyperPass essentially acts as a password manager that automatically inserts correct passwords. Unfortunately, it cannot know whether the program that is attempting to access the remote service is legitimate or malicious. Thus, an attacker with remote access to a user’s machine could successfully run a program on that machine which simply tries to log in to some service with the correct username, and HyperPass would naively fill in the correct password. As mentioned in § 3.2, we address this by displaying a user interface which demands confirmation from the user. If the user is not aware of trying to log in to the specified service (e.g., they didn’t just press Submit on a login form or open their IM program), they click “No” and the connection is denied. This user interface is not displayed by the User environment and interacting with it requires a user to be physically present – “possessing” the computer in an immediate physical way, and not simply in the very loose sense of a remote attacker being able to execute software on it.

4.4.2 Spying Spouses

While HyperPass’s Secure UI can assure that some user is physically in possession of the machine, it is not itself able to determine who this user is. Elsewhere in this paper, we assume that the user in physical possession is, in fact, the rightful owner of the services for which HyperPass is storing passwords. However, a “spying spouse” or another user who can gain physical access to the machine (and access to a user environment on it) will be able to convince HyperPass to log in as the machine’s true owner only using account names. Solutions to this are outside the scope of HyperPass. The only true solution is to safeguard physical access to the machine, though other solutions such as a master password or BIOS-based biometric protection are likely to be sufficient in practice.

5 Proof-of-Concept Implementation

5.1 Basic Framework

To evaluate the feasibility of our approach, we have developed a proof-of-concept implementation. HyperPass proposes two separate environments – a User environment (which runs the user’s unmodified operating system and applications) and a HyperPass environment (which runs the HyperPass code and stores passwords). The most straightforward way to implement this is using virtual machines. To provide these virtual machines, we ran HyperPass with both VirtualBox (using one VM for each environment) and with Xen (using a domU VM for the User environment and dom0 for HyperPass). We have tested both Windows and Linux User environments (mostly the latter). Our HyperPass environment was based on Linux. As we discuss in § 7, we expect the desire for a small Trusted Computing Base would motivate a real implementation of HyperPass to use a custom microvisor [13, 28] both to implement isolation and to support HyperPass’s Service Modules. However,
As described in § 3.3, Service Modules specify traffic of interest by domain name and TCP port, but Open vSwitch makes forwarding decisions based on header fields – that is, on IP addresses and not domain names. This issue is somewhat complicated by many services having domain names map to a fairly large and fluctuating pool of IP addresses. A straightforward way to have confronted this would have been to do a reverse DNS lookup at connection setup time, though this would have introduced considerably latency. Instead, we implemented a DNS Snooping component that instructs Open vSwitch to copy DNS traffic to HyperPass, and use this to build a map between IP addresses and domain names based on the same data that the User environment has seen. This allows us to match IP addresses to domain names at connection initialization time with a simple hash lookup.

Also worth noting is the introduction of POX [20]. POX is a framework for developing network control software using Python, and we built our proof-of-concept atop this framework. In particular, POX facilitates our programming of Open vSwitch via the OpenFlow [14] protocol and our parsing of DNS packets. The TLS components were implemented with the help of the pyOpenSSL library [22]. Support for changing service passwords through their web interfaces was implemented using the Python mechanize library which facilitates stateful programmatic web browsing [21], and our user interface was implemented using Python’s built-in TkInter package.

5.2 Service Modules

For HyperPass to be a workable solution, it must be able to support services that people actually use, which motivated us to implement Service Modules for a handful of services that we use regularly. We also wished to choose services that explored the generality of our approach. Specifically, we chose the following:

Web-based Services: For web-based services, we chose Facebook for its incredible popularity, and a favorite site of ours (and one we used for developing HyperPass) – GitHub. Implementing these Service Modules requires parsing HTTP requests and responses, and this accounts for much of their complexity. We confronted this by implementing a state-machine based library for parsing HTTP, which we reused for all web-based service modules; this library took approximately 200 lines of Python code – twice the length of our largest Service Module.

Instant Messaging: For an instant messaging protocol, we chose Microsoft Messenger. We discovered that while Microsoft Messenger uses a proprietary protocol for actual instant messaging and presence, its authentication is implemented using the same web service as hotmail.com, msn.com, and other Microsoft services – the same Service Module, in fact, allows access to all of these. Although we did not implement a Yahoo! Service Module, analysis of Yahoo! Messenger indicated that it works similarly.

Email: For an email service, we chose Gmail. While Gmail is typically known for its web-based interface, it also supports the IMAP and SMTP protocols which we wished to support. As Gmail is actually just a feature of iGoogle and Google+, we chose Google as our main service. For HyperPass to be a workable solution, it must be able to support services that people actually use, which motivated us to implement Service Modules for a handful of services that we use regularly. We also wished to choose services that explored the generality of our approach. Specifically, we chose the following:

Web-based Services: For web-based services, we chose Facebook for its incredible popularity, and a favorite site of ours (and one we used for developing HyperPass) – GitHub. Implementing these Service Modules requires parsing HTTP requests and responses, and this accounts for much of their complexity. We confronted this by implementing a state-machine based library for parsing HTTP, which we reused for all web-based service modules; this library took approximately 200 lines of Python code – twice the length of our largest Service Module.

Instant Messaging: For an instant messaging protocol, we chose Microsoft Messenger. We discovered that while Microsoft Messenger uses a proprietary protocol for actual instant messaging and presence, its authentication is implemented using the same web service as hotmail.com, msn.com, and other Microsoft services – the same Service Module, in fact, allows access to all of these. Although we did not implement a Yahoo! Service Module, analysis of Yahoo! Messenger indicated that it works similarly.

Email: For an email service, we chose Gmail. While Gmail is typically known for its web-based interface, it also supports the IMAP and SMTP protocols which we wished to support. As Gmail is actually just a feature of a “Google account”, we found that implementing a web-based Service Module for Google accounts allows access to not just Gmail, but also other Google features such as iGoogle and Google+. We additionally implemented two “slave” Service Modules to handle IMAP and SMTP. Unlike other Service Modules, these simply borrow their passwords from the main Google Service Module and do
not store or set passwords independently. The design of these Service Modules is quite simple compared to the web-based ones, as IMAP and SMTP are not only both easy-to-parse line-based protocols, but are stateful: they authenticate early on in the session (through the IMAP LOGIN [5] command and ESMTP AUTH PLAIN [16,17] mechanism respectively), after which we can simply pass traffic. Although we only used the SMTP and IMAP Service Modules with Gmail, we expect that they could be adapted for use with a number of different email providers relatively easily.

6 Performance Evaluation

Our implementation is only a proof-of-concept, and significant portions are written in a very high level language without particularly good performance characteristics. Nevertheless, we evaluate it here with an eye towards confirming that our approach does not incur such terrible overheads as to be completely unreasonable.

6.1 Microbenchmarks

We performed one microbenchmark on HyperPass – specifically, on our HTTP parsing library. As most of our Service Modules are for interacting with web-based services, and since all of these make use of our library for parsing the HTTP requests and responses, a performance problem with this code could be a major bottleneck. To test it, we captured a trace of approximately ten minutes of “ordinary” web browsing (accessing Facebook, Google, and so on), totalling 59MB. We ran this data through our parsing library, which completed in 0.163 seconds. As the best-case transfer time for this amount of data on a 30Mbps link is approximately 16 seconds, this is only an overhead of about 1%.

6.2 Macrobenchmarks

To evaluate HyperPass’s performance, we ran a number of tests both with HyperPass and from the “bare metal” environment of the machine – directly on its usual operating system with no hypervisor or virtualization involved (we later refer to this as the “Host” configuration). While comparing these two measurements can tell the total overhead involved in running our proof-of-concept, it would not tell the entire story. Our proof-of-concept implementation uses either Xen or VirtualBox to provide HyperPass’s two isolated environments (note that we only collected full measurements under VirtualBox, and it is those which we report on here). These are off-the-shelf hypervisors, and we expect that a real implementation of HyperPass would provide these two environments quite differently (we briefly explore this subject in §7), and would have significantly different performance attributes. Thus we also perform our tests in two other configurations. The first of these we call “VM”, and is simply our User environment VM running with VirtualBox’s default network configuration. The second (“Forwarding”) is very similar to the complete HyperPass configuration, except instead of programming Open vSwitch to send some traffic to HyperPass and allow HyperPass to modify it, we simply program it to forward traffic.

By comparing the VM configuration to the Host configuration, we can see what overhead virtualization alone adds. Significantly, we expect that this overhead is highly dependent on the exact virtualization mechanisms used and that it is not particularly meaningful in evaluating the feasibility of the HyperPass approach. By comparing between the VM and Forwarding configurations, we expect to see the additional overhead of the inter-VM networking and Open vSwitch. Finally, by comparing between the Forwarding and HyperPass configurations, we intend to measure the overhead of HyperPass’s core functionality – its flow classification and Service Modules.

HyperPass must intercept new TCP connections and classify them as either interesting (meaning they may contain authentication events that a Service Module will need to manipulate) or uninteresting (meaning they can simply be forwarded). We test the overhead of this classification process by measuring the time it takes to establish 100 connections for each of 10 different remote servers. Figure 5a shows that the virtualization and forwarding mechanisms add virtually no overhead, which is as expected since none of these actually perform the classification step. HyperPass adds an average of approximately 17ms per connection – an overhead of 25%. Though the overhead may seem to be high, this is just a one-time overhead incurred during flow setup. Additionally, it can be seen that HyperPass adds significant variation to the connection time compared to the negligible variation seen without HyperPass.

We also evaluate the latency and throughput of our fast path (uninteresting traffic) by performing tests with ping and iperf against a server on the same fast LAN as our test machine. Figure 5b shows that HyperPass adds approximately 1.6ms of latency. The throughput test (figure 5c) shows that all virtualized environments perform far worse than the bare metal environment. With respect to the second best throughput rate, HyperPass performs only marginally worse. This indicates that the bottleneck is not with HyperPass itself, but with VirtualBox’s network performance. We find this promising, as we believe future work on HyperPass will use a substantially different mechanism to separate the two environments (described further in §7). A counterintuitive result from this experiment is that the Forwarding configuration
<table>
<thead>
<tr>
<th></th>
<th>Bare metal environment hosting VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>User VM connected directly to the network</td>
</tr>
<tr>
<td>Forwarding</td>
<td>User VM connected to a VM which just forwards</td>
</tr>
<tr>
<td>HyperPass</td>
<td>User VM connected to HyperPass VM</td>
</tr>
</tbody>
</table>

Testing Configurations

(a) Connection Establishment Time

(b) Overall Network Latency

(c) Overall Network Throughput

(d) Google Account Login Time

Figure 5: HyperPass Evaluation
outperforms the VM configuration, despite the fact that Forwarding does everything that the VM configuration does and more. We repeated this experiment several times to convince ourselves that this result was, in fact, true. Our hypothesis is that it might be due to the VM configuration (which uses only a single VM) running the VM’s Linux networking stack and VirtualBox’s networking virtualization on the same CPU, while the two-VM Forwarding configuration runs the User environment VM’s Linux networking stack on one CPU while running Open vSwitch (considerably more lightweight than the full Linux network stack) and VirtualBox’s networking on a second VM/CPU.

Finally, we measure the overhead that HyperPass incurs when actually intercepting and altering authentication. This test was performed by disabling HyperPass’s login confirmation UI, and executing a script which actually goes through the entire process to log in to a Google Account as done from a browser. Figure 5d shows that HyperPass adds an overhead of 1.9 seconds for completing a login process. The HTTP Parsing microbenchmark indicates that the HTTP parsing involved is not a significant factor, so we conclude that this overhead is largely due to TCP connection establishment and TLS handshakes with both the client and the service. In our current implementation, the TCP and TLS initializations are done serially: first the connection to the service to make sure it connects and to validate the certificate, then the connection to the client. While serial ordering is logical, it may be possible (and worthwhile) to accelerate the process by performing some of these simultaneously. Moreover, it is worth noting that this overhead is only incurred during the login phase, and subsequent browsing is on the “fast path” and incurs minimal overhead.

7 Conclusion and Future Work

HyperPass is a hypervisor-based approach for securing online service passwords against theft from malicious software, as well as from some of the network and phishing attacks. Through a proof-of-concept implementation, we show that the approach is workable. While this proof-of-concept is far from a fully realized system, early benchmarks indicate that the overhead the system incurs is not unreasonable.

The HyperPass proof-of-concept is implemented using a relatively small amount of custom code and a combination of relatively large off-the-shelf components: VirtualBox, Xen, Linux, Open vSwitch, Python, and POX. This results in a significant TCB, only a small portion of which is truly dedicated to implementing HyperPass. One direction for future work revolves around reducing the size of this TCB. Specifically, we think the right solution is the implementation of a custom “microvisor” with exactly the required functionality – an approach typified by CloudTerminal.

We also see the possibility for HyperPass to provide additional security benefits. For example, HyperPass already obviates the need for a user to remember his or her password; given this, there is little reason that HyperPass could not change the password regularly. Such a policy would prevent passwords stolen from techniques that we do not protect against (such as compromise of the service) from being replayable.

HyperPass’s main method of protecting passwords from malicious software is through keeping passwords in an isolated environment and merging passwords with normal traffic at the network level. However, many of its benefits also come from exchanging the knowledge factor of passwords for the possession factor of having physical access to the machine running HyperPass. A consequence of this exchange is that authentication is now tied to HyperPass, which is unfortunate as users may wish to access accounts from multiple machines – especially with the increasing prevalence of smartphones and tablets. Although features like Google’s “application-specific passwords” provide some remedy for this by allowing one to generate a separate password for use by other devices, this type of feature is not available for all services. Future work in HyperPass may explore other solutions, perhaps by creating a smartphone version of HyperPass.

A key property of HyperPass is that it requires service-specific support modules. To be useful, it must contain support for services that people use, and this support must be kept up-to-date – a potentially tricky task in the face of quickly evolving online services. This clearly motivates the development of a secure update mechanism for HyperPass to deploy new and updated service modules. Although there are far too many online services for us to implement support for all of them ourselves, we believe that a community effort dedicated to the implementation, maintenance, and auditing of service modules could address the needs of many users. Furthermore, we feel confident that such efforts are tractable, as they are similar to the efforts of other successful community-oriented projects, such as the Debian Project [6].

References

