TrustBase: An Architecture to Repair and Strengthen Certificate-based Authentication

Mark O'Neill, Scott Heidbrink, Scott Ruoti, Jordan Whitehead, Dan Bunker, Luke Dickinson, Travis Hendershot, Joshua Reynolds, Kent Seamons, and Daniel Zappala, ***Brigham Young University***

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/oneill

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

TrustBase: An Architecture to Repair and Strengthen Certificate-based Authentication

Mark O'Neill Scott Heidbrink Scott Ruoti Jordan Whitehead Dan Bunker
Luke Dickinson Travis Hendershot Joshua Reynolds
Kent Seamons Daniel Zappala
Brigham Young University
mto@byu.edu, sheidbri@byu.edu, ruoti@isrl.byu.edu, jaw@byu.edu, dbunked@gmail.com
luke@isrl.byu.edu, tmanhendy@isrl.byu.edu, joshua@isrl.byu.edu
seamons@cs.byu.edu, zappala@cs.byu.edu

Abstract

The current state of certificate-based authentication is messy, with broken authentication in applications and proxies, along with serious flaws in the CA system. To solve these problems, we design TrustBase, an architecture that provides certificate-based authentication as an operating system service, with system administrator control over authentication policy. TrustBase transparently enforces best practices for certificate validation on all applications, while also providing a variety of authentication services to strengthen the CA system. We describe a research prototype of TrustBase for Linux, which uses a loadable kernel module to intercept traffic in the socket layer, then consults a userspace policy engine to evaluate certificate validity using a variety of plugins. We evaluate the security of TrustBase, including a threat analysis, application coverage, and hardening of the Linux prototype. We also describe prototypes of TrustBase for Android and Windows, illustrating the generality of our approach. We show that TrustBase has negligible overhead and universal compatibility with applications. We demonstrate its utility by describing eight authentication services that extend CA hardening to all applications.

1 Introduction

Server authentication on the Internet currently relies on the certificate authority (CA) system to provide assurance that the client is connected to a legitimate server and not one controlled by an attacker. Unfortunately, certificate validation is challenged by significant problems. First, applications frequently do not properly validate the server's certificate [20, 17, 5, 35]. This is caused by failure to use validation functions, incorrect usage of libraries, and also developers who disable validation during development and forget to enable it upon release. Second, TLS interception, used by numerous firewall appliances and software (as well as malware), compromises the integrity of end-to-end encryption [24, 34], with many firewalls having significant implementation bugs that break authentication [7, 11]. Third, the CA system itself is vulnerable to being hijacked even when applications and proxies are implemented correctly. This is largely due to the fact that most CAs are able to sign certificates for any host, reducing the strength of the CA system to that of the weakest CA [12]. This weakness was exploited in the 2011 DigiNotar hack [25], and is exacerbated by CAs that do not follow best practices [31, 10] and by governmental ownership and access to CAs [13, 40].

Due to these problems, there are a number of recent proposals to improve or replace the current CA trust model. These include multi-path probing [31, 41, 1, 23] or other systems that vouch for the authenticity of a certificate [15, 2, 3], DNS-based authentication [21], certificate pinning [16, 32], and audit logs [27, 39, 14, 26]. Unfortunately, the majority of applications have not yet integrated these improvements. Even relatively simple fixes, such as certificate revocation, are beset with problems [29]. The result is that there is no de facto standard regarding where and how certificate validation should occur, and it is currently spread between applications, TLS libraries, and interception proxies [11].

Several projects have tried to address these issues, fixing broken authentication in existing applications, while providing a means to deploy improved authentication services. Primary among these is CertShim, which uses the `LD_PRELOAD` environment variable to replace functions in dynamically-loaded security libraries [4]. However, this approach does not provide universal coverage of all existing applications, does not provide administrators singular control over certificate authentication practices, does not protect against several important attacks, and has significant maintenance issues. Fahl takes a different approach that rewrites the library used for authentication by Android applications [18], while also including pluggable authentication modules. This approach is well-suited for Android because all applications are written in Java, but

it is difficult to extend this approach to operating systems that provide more general programming language support.

In this paper, we explore a different avenue for fixing these problems by centralizing authentication as an operating system (OS) service and giving system administrators and OS vendors control over authentication policy. These two motivating principles result in TrustBase, an architecture for certificate authentication that secures existing applications, provides simple deployment of improved authentication services, and overcomes shortcomings of previous approaches. TrustBase provides universal coverage of existing applications, supports both direct and opportunistic TLS[1], is hardened against unprivileged local adversaries, is supported on both mobile and desktop operating systems, and has negligible overhead.

To centralize authentication as an operating system service, TrustBase uses a combination of traffic interception to harden certificate validation for existing applications and a validation API to simplify authentication for new or modified applications. TrustBase intercepts network traffic between the socket layer and the transport layer, where it detects the initiation of TLS connections, extracts handshake information, validates the server's certificate using a variety of configurable authentication services, and then allows or blocks the connection based on the results of this additional validation. This allows TrustBase to harden certificate validation in an application-agnostic fashion, irrespective of what TLS library is employed. TrustBase also includes a simple certificate validation API that applications call directly, which extends authentication services to new or modified applications, while also providing compatibility with TLS 1.3.

To provide system administrator control, TrustBase provides a policy engine that enables an administrator to choose how certificate authentication is performed on the host, with a variety of authentication services that can be used to harden the CA system. The checks performed by authentication services are complementary to any existing certificate validation performed by applications. This approach both protects against insecure applications and transparently enables existing applications to be strengthened against failures of the CA system. For example, a browser that validates the extended validation (EV) certificate of a bank is doing the best it currently can, but it is still vulnerable to a compromised CA, allowing a man-in-the-middle (MITM) to present fake but valid certificates. One possible use of TrustBase is to configure the use of notaries that check whether hosts across the Internet are exposed to the same certificate for the bank.[2]

[1] Opportunistic TLS is TLS initiated via an optional upgrade from a plaintext protocol.

[2] Keys for notaries can be pinned in advance, so they are not vulnerable to the MITM.

TrustBase enables system administrators and OS vendors to enforce a number of policies regarding TLS. For example, an administrator could require revocation status checking, disallow weak cipher suites, or mandate that Certificate Transparency be used to protect against active man-in-the-middle (MITM) attacks. An OS vendor could ship TrustBase with strong default protections against broken applications, such as enforcing best practices for validating a certificate chain, requiring hostname validation, and pinning certificates for the most popular web sites and applications. As TLS becomes more widespread, TrustBase could easily be extended to provide the capability to report on the use of all applications that do *not* use TLS, so that an organization could better manage or even block insecure applications. All of these improvements can be made without requiring user interaction or configuration.

Our contributions include:

- **An architecture for certificate validation that prioritizes operating system centralization and system administrator control:** TrustBase offers standard certificate validation procedures and optionally adds additional authentication services, both of which are enforced by the operating system and controlled by the administrator or OS vendor. This repairs broken validation for poorly-written applications and can strengthen the validation done by all applications. TrustBase provides a policy engine that enables an administrator to use policies that define how multiple authentication services cooperate, for example using unanimous consent or threshold voting.

- **A research prototype of TrustBase:** We develop a loadable kernel module that provides general traffic interception and TLS handling for Linux. This module communicates via the Netlink API to the policy engine residing in user space for parsing and validation of certificates. We describe how this same architecture can be implemented on other operating systems and give details of our current Android and Windows versions. We provide source code and developer documentation for our prototypes, with licensing for both commercial and non-commercial purposes.

- **A security analysis of TrustBase:** We provide a security analysis of TrustBase, including its centralization, application coverage, and the hardening we have done on the Linux implementation. We describe a threat analysis and demonstrate how TrustBase can thwart attacks that include a hacked CA, a subverted local root store, and a STARTTLS downgrade attack. We also demonstrate the ability of

TrustBase to fix applications that do not validate hostnames or skip certificate validation altogether.

- **An evaluation of TrustBase:** We evaluate the TrustBase prototype for performance, compatibility, and utility. (1) We show that TrustBase has negligible performance overhead, with no measurable impact on latency. (2) We demonstrate that TrustBase enforces correct certificate validation on all popular Linux libraries and tools and on the most popular Android applications. (3) We describe eight authentication services that we have developed and report on how simple and straightforward it was to develop these services for TrustBase.

2 Related Work

Three systems aim to tackle similar problems as TrustBase.

Fahl et al. proposed a new framework for Android applications that would help developers correctly use TLS [18]. Their work follows a similar principle to ours—instead of letting developers implement their own certificate validation code, validation is a service, and it incorporates a pluggable framework for authentication schemes. Fahl's approach is well-suited to mobile operating systems such as Android, where all applications are written in Java, but it is difficult to extend this approach to operating systems that provide more general programming language support.

Another Android system, MITHYS, was developed to protect Android applications from MITM attacks [6]. It first attempts to MITM applications that establish TLS connections and, if successful, continues using the MITM and provides certificate validation using a notary service hosted in the cloud. MITHYS only works for HTTPS traffic, adds significant delays to all TLS connections that it protects (one to ten seconds), and only supports the current CA system.

The most closely related system to TrustBase is CertShim [4]. Like TrustBase, CertShim is an attempt to immediately fix TLS problems in existing apps and also support new authentication services. CertShim works by utilizing the `LD_PRELOAD` environment variable to replace functions in dynamically-loaded security libraries with their own wrappers for those functions. This method has an advantage over TrustBase in that CertShim does not need to perform double validation for cases where an application is already performing certificate validation correctly. Because TrustBase uses traffic interception to enforce proper certificate validation, its checks are in addition to what applications may already do (either correctly or incorrectly). In addition, CertShim's wrapping of validation functions means that it can more easily override the CA system in the case where administrators want an application to accept alternative certificates, though this will only work with applications that CertShim supports and that do not validate against hard-coded certificates or keys.

TrustBase has advantages that set it apart from CertShim in several notable ways:

(1) **Coverage.** TrustBase intercepts all secure traffic and thus can independently validate certificates for all applications, regardless of what library they used, how they were compiled, what user ran them, or how they were spawned. CertShim does not support browsers, and it cannot perform validation for applications in all scenarios. For example, applications using custom or unsupported security libraries (e.g., BoringSSL, NSS, MatrixSSL, more-recent GnuTLS, etc.), applications statically linked with any security library, and applications spawned without being passed CertShim's path in the `LD_PRELOAD` environment variable (e.g., spawned by `execv` or spawned by a user without that environment setting) will not have their certificates validated by CertShim.

(2) **Maintenance.** TrustBase only needs to maintain compatibility with the TLS specification and the signatures of high-level functions of TCP in the Linux kernel. As a datapoint, the latter has had only two minor changes since Linux 2.2 (released 1999)—one change was to add a parameter, the other was to remove it. In contrast, CertShim relies on data structures internal to the security libraries it supports, and libraries change their internals with surprising frequency. The current versions of PolarSSL (now mbed TLS) and GnuTLS were no longer compatible with CertShim, one year after its release.

(3) **Administrator Control.** TrustBase ensures that only system administrators can load, unload, bypass, or modify its functionality, so that every secure application is subject to its configured policies. With CertShim, guest users and applications can easily opt out of its security policies by removing CertShim from their `LD_PRELOAD` environment variable, and developers can bypass CertShim by statically-linking with security libraries, using an unsupported TLS library, or spawning child processes without CertShim in their environment.

(4) **Local Adversary Protection.** TrustBase uses a trust model that protects against a local adversary, wherein a nonprivileged, local, malicious application attempts to bypass or alter certificate validation. Recent studies of TLS MITM behavior suggest that local malware acting as a MITM is more prevalent than remote MITM attackers [24, 34]. TrustBase protects against this case by using a protected Netlink protocol, privileged policy engine, protected files, and kernel module that cannot be removed by a nonprivileged user. CertShim's attack model does not address this case. In fact, malware uses the same `LD_PRELOAD` mechanism [28].

(5) **Opportunistic TLS Enforcement.** TrustBase can enforce the use of TLS in plaintext protocols that optionally allow upgrades to TLS, such as STARTTLS, significantly reducing the attack surface for downgrade attacks. Since CertShim hooks into TLS library calls, it cannot be invoked if no calls occur.

3 TrustBase

TrustBase is motivated by the need to fix broken authentication in applications and strengthen the CA system, using the two motivating principles that authentication should be centralized in the operating system and system administrators should be in control of authentication policies on their machines. In this section, we discuss the threat model, design goals, and architecture of the system.

3.1 Threat Model

In our threat model, an active attacker attempts to impersonate a remote host by providing a fake certificate. Our attacker includes remote hosts as well as MITM attackers located anywhere along the path to a remote host. The goal of the attacker is to establish a secure connection with the client.

The application under attack may accept the fake certificate for the following reasons:

- The application employs incorrect certificate validation procedures (e.g., limited or no validation) and the attacker exploits his knowledge of this to trick the application into accepting his fake certificate.
- The attacker or malware managed to place a rogue certificate authority into the user's root store (or another trust store used by the application) so that he has become a trusted certificate authority. The fake certificate authority's private key was then used to generate the fake certificate used in the attack.
- Non-privileged malware has altered or hooked security libraries the application uses to force acceptance of fake certificates (e.g., via malicious OpenSSL hooks and LD_PRELOAD).
- A legitimate certificate authority was compromised or coerced into issuing the fake certificate to the attacker.

Local attackers (malware) with root privilege are outside the scope of our threat model. In addition, we consider only certificates from TLS connections directly made from the local system to a designated host, and not those that may be present in streams higher up in the OSI stack or indirectly from other hosts or proxies via protocols like onion routing.

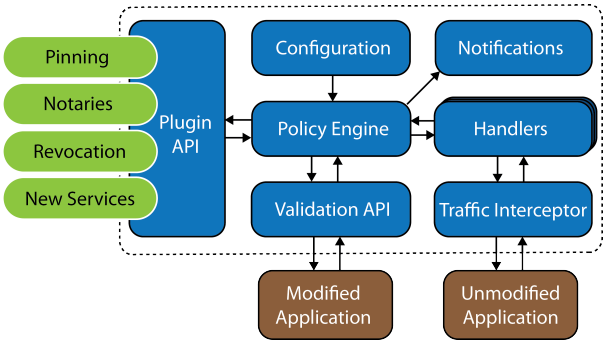

Figure 1: TrustBase architecture overview

3.2 Design Goals

The design goals for TrustBase are: **(1) Secure existing applications.** TrustBase should override incorrect or absent validation of certificates received via TLS in current applications. **(2) Strengthen the CA system.** TrustBase should provide simple deployment of authentication services that strengthen the validation provided by the CA system. **(3) Full application coverage.** All incoming certificates should be validated by TrustBase, including those provided to both existing applications and future applications. However, this does not include certificates from connections not made directly by the system, such as certificates delivered through onion routing. **(4) Universal deployment.** The TrustBase architecture should be designed to work on any major operating system, including both desktop and mobile platforms. **(5) Negligible overhead.** TrustBase should have negligible performance overhead. This includes ensuring that the user experience for applications is not affected in any way, except when TrustBase prevents an application from establishing an insecure connection.

3.3 Architecture

The architecture for TrustBase is given in Figure 1. These components are described below:

3.3.1 Traffic Interceptor

The traffic interceptor intercepts all network traffic and delivers it to registered handlers for further processing. The interceptor is generic, lightweight, and can provide traffic to any type of handler. Traffic for any specific stream is intercepted only as long as a handler is interested in it. Otherwise, traffic is routed normally.

The traffic interceptor is needed to secure existing applications. If a developer is willing to modify her application to call TrustBase directly for certificate validation, then she can use the validation API. Administrators can con-

figure TrustBase to not intercept traffic from applications using this API.

3.3.2 Handlers

Handlers are state machines that examine a traffic stream to isolate data used for authenticating connections and then pass this data to the policy engine. Data provided to the policy engine includes everything from the relevant protocol that is intercepted.[3] For example, with TLS this includes the `ClientHello` and `ServerHello` data in addition to the server certificate chain and the server hostname. The handler will allow or abort the connection, based on the policy engine's response.

TrustBase currently has both a TLS handler and an opportunistic TLS handler (e.g., STARTTLS), and due to the design of the traffic interceptor it is easy to add support for new secure transport protocols as they become popular (e.g., QUIC, DTLS).

3.3.3 Policy Engine

The policy engine is responsible for using the registered authentication services to validate the server certificate extracted by the handler. The policy engine also aggregates validation responses if there are multiple active authentication plugins. The policy is configured by the system administrator, with sensible operating system defaults for ordinary users.

When the policy engine receives a validation request from a handler, it will query each of the registered authentication services to validate the server's certificate chain and host data. Authentication services can respond to this query in one of four ways: *valid*, *invalid*, *abstain*, or *error*. Abstain and error responses are mapped to the valid or invalid responses, as defined in a configuration file.

To render a decision, the policy engine classifies plugins as either "necessary" or "voting", as defined in the configuration file. All plugins in the "necessary" category must indicate the certificate is valid, otherwise the policy engine will mark the certificate as invalid. If the necessary plugins validate a certificate, the responses from the remaining "voting" plugins are tallied. If the aggregation of valid votes is above a preconfigured threshold, the certificate is deemed valid by the policy engine. A write-protected configuration file lists the plugins to load, assigns each plugin to an aggregation group ("necessary" or "voting"), defines the timeout for plugins, etc.

3.3.4 Plugin API

TrustBase defines a robust plugin API that allows a variety of authentication services to be used with TrustBase. The policy engine queries each authentication service by supplying host data and a certificate chain, and the authentication service returns a response. We provide both an asynchronous plugin API and a synchronous plugin API to facilitate the needs of different designs.

The synchronous plugin API is intended for use by simple authentication methodologies. Plugins using this API may optionally implement `initialize` and `finalize` functions for any setup and cleanup they need to perform. For example, a plugin may want to store a cache or socket descriptor for long-term use during runtime. Each plugin must also implement a `query` function, which is passed a data object containing a query ID, hostname, IP address, port, certificate chain, and other relevant context. The certificate chain is provided to the plugin DER encoded and in openssl's `STACK_OF(X509)` format for convenience. The `query` function returns the result of the plugin's validation of the query data (valid, invalid, abstain, or error) back to the policy engine.

The asynchronous plugin API allows for easier integration with more advanced designs, such as multithreaded and event-driven architectures. This API supplies a callback function through the `initialize` function that plugins must use to report validation decisions, using the query ID supplied by the data supplied to `query`. Thus the `initialize` function is required so that plugins may obtain the callback pointer (the `finalize` function is still optional). Asynchronous plugins also implement the `query` function, but return a status code from this function immediately and instead report their validation decision using the supplied callback.

3.3.5 Validation API

The validation API provides a direct interface to the policy engine for certificate validation. New or modified applications can use this API to simplify validation, avoid common developer mistakes, and take advantage of TrustBase authentication services. Applications can use the API to validate certificates or request pinning for a self-signed certificate. The API also allows the application to receive validation error messages from TrustBase, allowing it to display errors directly in the application (TrustBase displays notifications through the operating system).

3.4 Addressing Certificate Pinning

Some applications have implemented certificate pinning to provide greater security in cases where the hosts that the application visits are static and known, rather than using the CA system for certificate validation. TrustBase wants to avoid the situation where its authentication services declare a certificate to be invalid when the application has validated it with pinning, but should also adhere to its core

[3] This enables plugins to provide authentication methods that utilize TLS extensions and cipher suite information.

tenant that the system administrator should have ultimate control over how certificates are validated. Our measurements indicate that this circumstance is rare and affects relatively few applications, since the problem only arises when a certificate offered by a host does not also validate by the CA system (e.g., a self-signed certificate). In the short term, TrustBase solves this problem by using the configuration file to allow whitelisting of programs that should bypass TrustBase's default policies. In the long term, this problem is solved by applications migrating to the validation API.

3.5 TLS 1.3 and Overriding the CA System

There are two situations where TrustBase cannot use default traffic interception to accomplish its primary goals. First, when an application uses TLS 1.3, the certificates that are exchanged are encrypted, preventing TrustBase from using passive traffic interception to independently validate certificates. Second, in some cases a system administrator may want to distrust the CA system entirely and rely solely on alternative authentication services. For example, the administrator may want to force applications currently using CA validation to accept self-signed certificates that have been validated using a notary system such as Convergence[31], or she may want to use DANE[21] with trust anchors that differ from those shipped with the system. When this occurs, TrustBase will use the new authentication service and determine the certificate is valid and allow a connection as configured by the administrator, but applications using the CA system may reject the certificate and terminate the connection. We stress that such a policy would not be intended to override strong certificate checks done by a browser (e.g., when communicating with a bank), but to provide a path for migrating away from the CA system as stronger alternatives emerge.

To handle both TLS 1.3 and overriding the CA system, TrustBase provides two options. The preferred option is to modify applications to rely on TrustBase for certificate validation, rather than performing their own checks. This is facilitated by the validation API described above. This enables new or modified applications to use the full set of authentication services provided by TrustBase in a natural manner.

A second option is to employ a local TLS proxy that can coerce existing applications that rely on the CA system to use new authentication services instead. The use of a proxy also allows TrustBase plaintext access to the server's certificate under TLS 1.3. TrustBase gives the administrator the option of running such a proxy, but it is activated only in those cases where it is needed, namely when the policy engine determines a certificate is valid but the CA system would reject it. The proxy employed is a modified fork of `sslsplit` [38] and has shown itself to be scalable and performant in our experimentation. Note that in most cases this is not needed—for example, under Convergence, the certificates validated by notaries would likely also be validated by the CA system unless the certificate was self-signed, which is a situation likely to exist until CA alternatives gain significant traction. Given the vulnerabilities noted recently with proxies [11] administrators should exercise caution using this feature. Due to the features of the Windows root store, TrustBase on Windows can override the CA system without the use of a local proxy, as explained in Section 6.4.

3.6 Operating System Support

We designed the TrustBase architecture so that it could be implemented on additional operating systems. The main component that may need to be customized for each operating system is the traffic interception module. We are optimistic that this is possible because the TCP/IP networking stack and sockets are used by most operating systems.

Our Linux implementation is described in the following section. We also have a working prototype of TrustBase for Windows, which uses the Windows Filtering Platform API.

Mac OSX provides a native interface for traffic interception between the TCP and socket levels of the operating system. Apple's Network Kernel Extensions suite provides a "Socket Filter" API that could be used as the traffic interceptor.

For iOS, Apple provides a Network Extension framework that includes a variety of APIs for different kinds of traffic interception. The App Proxy Provider API allows developers to create a custom transparent proxy that captures application network data. Also available is the Filter Data Provider API that allows examination of network data with built-in "pass/block" functionality.

Because Android uses a variant of the Linux kernel, we believe our Linux implementation could be ported to Android with relative ease. We have a prototype of TrustBase on Android that instead uses the `VPNService` to intercept traffic.

4 Linux Implementation

We have designed and built a research prototype of TrustBase for Linux. The source code is available at `owntrust.org`.

We have developed a loadable kernel module (LKM) to intercept traffic at the socket layer, as data transits between the application and TCP handling kernel code. No modification of native kernel code is required, and the LKM can be loaded and unloaded at runtime. Similarly to

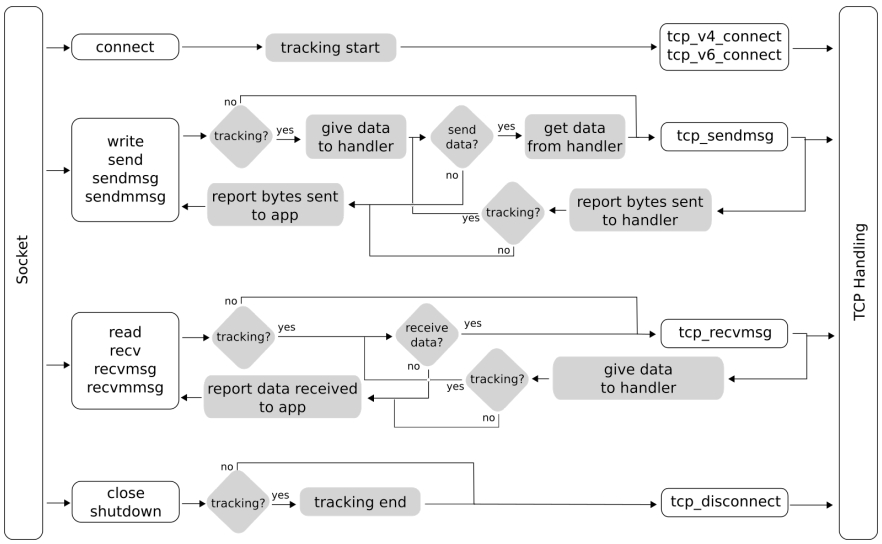

Figure 2: Linux Traffic Interceptor simplified flowchart. Grey boxes correspond to hooks for handlers, white boxes are native system calls and kernel functions

how Netfilter operates at the IP layer, TrustBase can intercept traffic at the socket layer, before data is delivered for TCP handling, and pass it to application-level programs, where it can be (optionally) modified and then passed back to the native kernel code for delivery to the original TCP functionality. Likewise, interception can occur after data finishes TCP processing and before it is delivered to the application. This enables TrustBase to efficiently intercept TLS connections in the operating system and validate certificates in the application layer.

The following discussion highlights the salient features of our implementation.

4.1 Traffic Interceptor

TrustBase provides generic traffic interception by capturing traffic between sockets and the TCP protocol. This is done by hooking several kernel functions and wrapping them to add traffic interception as needed. An overview of which functions are hooked and how they are modified is given in Figure 2. Items in white boxes on the left side of the figure are system calls. Items in white boxes on the right side of the figure are the wrapped kernel functions. The additional logic added to the native flow of the kernel is shown by the arrows and gray boxes in Figure 2.

When the TrustBase LKM is loaded, it hooks into the native TCP kernel functions whose pointers are stored in the global kernel structures `tcp_prot` (for IPv4) and `tcpv6_prot` (for IPv6). When a user program invokes a system call to create a socket, the function pointers within the corresponding protocol structure are copied into the newly-created kernel socket structure, allowing different protocols (TCP, UDP, TCP over IPv6, etc.) to be invoked by the same common socket API. The function pointers in the protocol structures correspond to basic socket operations such as sending and receiving data, and creating and closing connections. Application calls to `read`, `write`, `sendmsg`, and other system calls on that socket then use those protocol functions to carry out their operations within the kernel. Note that within the kernel, all socket-reading system calls (`read`, `recv`, `recvmsg`, and `recvmmsg`) eventually call the `recvmsg` function provided by the protocol structure. The same is true for the corresponding socket write system calls, as each result in calling the kernel `sendmsg` function. When the LKM is unloaded, the original TCP functionality is restored in a safe manner.

From top to bottom in Figure 2, the functionality of the traffic interceptor is as follows. First, a call to `connect` informs the handler that a new connection has been created, and the handler can choose to intercept its traffic.

Second, when an application makes a system call to send data on the socket, the interceptor checks with the handler to determine if it is tracking that connection. If so, it forwards the data to the handler for analysis, and the handler chooses what data (potentially modified by the handler), if any, to relay to native TCP-sending code. After attempting to send data, the interceptor informs the handler how much of that data was successfully placed into the kernel's send buffer and provides notification of any errors that occurred. At this point the interceptor allows the handler to send additional data, if desired. This process continues until the handler indicates it no longer wishes to send data. The interceptor then queries the handler for the return values it wishes to report to the

application (such as how many bytes were successfully sent or an error value) and these values are returned to the application.

Third, a similar, reversed process is followed for the reception of data from the network. If the interceptor is tracking the connection it can choose whether to receive data processed by TCP handling. Any data received is reported to the handler, which can choose whether to report a different value to the application. Note that handlers are allowed to report arbitrary values to applications for the amount of data sent or received, including false values, to allow greater flexibility in connection handling, or to maintain application integrity when injecting additional bytes into a stream. For example, to provide more time to obtain and parse a message, a handler may indicate to an application that zero bytes have been received on a nonblocking socket, even though some or all of the data may have already been received. After the handler has completed its operation it can report to a subsequent receive call from the application that bytes were received, and fill the application's provided buffer with relevant data. As another example, if a handler wishes to append data to a message successfully transferred to the OS by an application using the `send` system call, it should enforce that the return value of this function be the number of bytes the application expects to have been sent, rather than a higher number that includes the added bytes.

Finally, a call to `close` (on the last remaining socket descriptor for a connection) or `shutdown` informs the handler that the connection is closed. Note that the handler may also choose to abandon tracking of connections before this point.

Handlers for various network observation and modification can be constructed by implementing a small number of functions, which will be invoked by the traffic interceptor at runtime. These functions roughly correspond to the grey boxes in Figure 2. For example, handlers must implement functions to send and receive data, indicate whether to continue or cease tracking of a connection, etc. The traffic interceptor calls these functions to provide the handler with data, receive data from the handler to be forwarded to applications or remote hosts, and other tasks. Such an architecture allows developers to implement arbitrary protocol handlers as simple finite state machines, as demonstrated by the TLS handler and opportunistic TLS handlers described in the following subsections.

Another option for implementing traffic interception would have been to use the Netfilter framework, but this is not an optimal approach. TrustBase relies on parsing traffic at the application layer, but Netfilter intercepts traffic at the IP layer. For TrustBase to be implemented using Netfilter, TrustBase would need to transform IP packets into application payloads. This could be done either by implementing significant portions of TCP, including out-of-order handling and associated buffers, or passing traffic through the network stack twice, once to parse the IP packets for TrustBase and once for forwarding the traffic to the application. Both of these options are problematic, creating development and performance overhead, respectively.

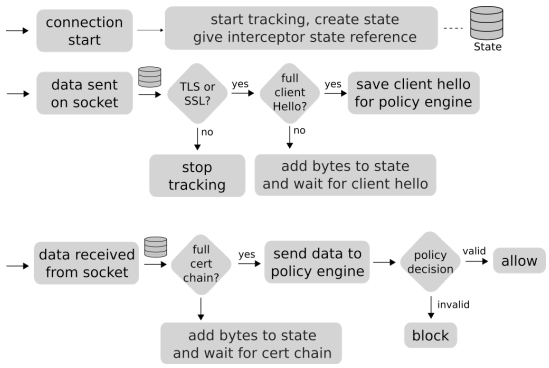

Figure 3: Simplified view of TLS handler

4.2 TLS Handler

TrustBase includes a handler for the traffic interceptor dubbed the "TLS handler". The TLS handler extracts certificates from TLS network flows and forwards them to the policy engine for validation.

Figure 3 provides a high-level overview of how this handler operates. When a new socket is created, the handler creates state to track the connection, which the handler will have access to for all subsequent interactions with the interceptor. The destination IP address and port of the connection and PID of the application owning the connection are provided to the handler during connection establishment by the interceptor. Since the handler is implemented in a LKM, the PID of the socket can be used to obtain any further information about the application such as the command used to run it, its location, and even memory contents.

When data is sent on the socket, the handler checks state data to determine whether the connection has initiated a TLS handshake. If so, then it expects to receive a `ClientHello`; the handler saves this message for the policy engine so that it can obtain the hostname of the desired remote host, if the message contains a Server Name Indication (SNI) extension. If SNI is not used, a log of applications' DNS lookups can be used to infer the intended host,[4] similar to work by Bates et al. [4]

When data is received on the socket, the TLS handler waits until it has received the full certificate chain, then

[4]Our experimentation showed that all popular TLS implementations and libraries now use SNI, and Akamai reports HTTPS SNI global usage at over 98% [33], so this fallback mechanism is almost never needed.

it sends this chain and other data to the policy engine for parsing and validation.

Note, the TLS handler understands the TLS record and handshake protocols but does not perform interpretations of contained data. This minimizes additions to kernel-level code and allows ASN.1 and other parsing to be done in userspace by the policy engine.

4.3 Opportunistic TLS Handler

We have also implemented an opportunistic TLS handler, which provides TrustBase support for plaintext protocols that may choose to upgrade to TLS opportunistically, such as STARTTLS. This handler performs passive monitoring of plaintext protocols (e.g., SMTP), allowing network data to be fast-tracked to and from the application and does not store or aggressively process any transiting data. If at some point the application requests to initiate a TLS connection with the server (e.g., via a STARTTLS message), the handler processes acknowledgments from the server and then delivers control of the connection to the normal TLS handler, which is free to handle the connection as if it were conducting regular TLS.

It should be noted that the use of opportunistic TLS protocols by applications is subject to active attackers who perform stripping attacks to make the client believe the server does not support TLS upgrades, an existing vulnerability well documented by recent work [9, 19, 22]. TrustBase can prevent this type of attack, as discussed in Section 5.

4.4 Policy Engine

The policy engine receives raw certificate data from the TLS handler and then validates the certificates using the configured authentication services. To avoid vulnerabilities that may arise from performing parsing and modification of certificates in the kernel, all such operations are carried out in user space by the policy engine.

Communication between TrustBase kernel space and user space components is conducted via Netlink, a robust and efficient method of transferring data between kernel and user space, provided natively by the Linux kernel. The policy engine asynchronously handles requests from the kernel module, freeing up the kernel threads to handle other connections while a response is constructed.

Native plugins must be written in either C or C++ and compiled as a shared object for use by the policy engine. However, in addition to the plugin API, TrustBase supports an addon API that allows plugins to be written in additional languages. Addons provide the code needed to interface between the native C of the policy engine and the target language it supports. We have implemented an addon to support the Python language and have created several Python plugins.

5 Security Analysis

The TrustBase architecture, prototype implementation, and sample plugins have many implications for system security. In this section we provide a security analysis of the centralized system design, application coverage, protection of applications from attackers, and protection of TrustBase itself from attackers.

5.1 Centralization

Concentrating certificate validation in an operating system service has some risks and benefits. Any vulnerability in the service has the potential to impact all applications on the system. An exploit that grants an attacker root permission leads to compromise of the host. An exploit that causes a certificate to be rejected when it should be accepted is a type of denial-of-service attack. We note that if an attacker is able to get TrustBase to accept a certificate when it should not, any application that does its own certificate authentication correctly will be unaffected. If the application is broken, the TrustBase failure will not make the situation any worse than it already was. The net effect is a lost opportunity to make it better.

The risks of centralization are common to any operating system service. However, centralization also has a compelling upside. For instance, all of our collective effort can be centered on making the design and implementation correct, and all applications can benefit.[5] Securing a single service is more scalable than requiring developers to secure each application or library independently. It also enforces an administrator's preferences regardless of application behavior. Additionally, when a protocol flaw is discovered, it can be more rapidly tested and patched, compared to having to patch a large number of applications.

5.2 Coverage

Since one of the goals of TrustBase is to enforce proper certificate validation on all applications on a system, the traffic interceptor is designed to stand between the transport and application layers of the OS so that it can intercept and access all TLS flows from local applications. The handlers associated with the traffic interception component are made aware of a connection when a `connect` call is issued and can associate that connection with all data flowing through it. Applications that utilize their own

[5] All applications would likewise benefit from caching among authentication services.

custom TCP/IP stack must utilize raw sockets, which require administrator privileges and are therefore implicitly trusted by TrustBase.

To obtain complete coverage of TLS, our handlers need only monitor initial TLS handshakes (standard TLS) and the brief data preceding them (STARTTLS). The characteristics of TLS renegotiation and session termination are compatible with our approach.

In TLS renegotiation, subsequent handshakes use key material derived using the master secret of the first handshake. Thus if the policy engine correctly authenticates and validates the first handshake, TLS renegotiations are implicitly verified as well. Attackers who obtained sufficient secrets to trigger a renegotiation, through some other attack on the TLS protocol or implementation (outside our threat model), have no need to take advantage of renegotiation as they have complete control over the connection already. We also note that renegotiation is rare and typically used for client authentication for an already authenticated server, and has become less relevant for SGC or refreshing keys [37].

Session termination policies for TLS allow us to associate each TLS session with only one TCP connection. In TLS, a close notify must be immediately succeeded by a responding close notification and a close down of the connection [8]. Subsequent reconnects to the target host for additional TLS communication are detected by the TrustBase traffic interceptor and presented to the handlers. We have found that TLS libraries and applications do indeed terminate a TCP session when ending a TLS session, although many of them fail to send an explicit TLS close notification and rely solely on TCP termination to notify the remote host of the session termination.

5.3 Threat Analysis

The coverage of TrustBase enables it to enforce both proper and additional certificate validation procedures on TLS-using applications. There are a variety of ways that attackers may try perform a TLS MITM against these applications. A selection of these methods and discussion of how TrustBase can protect against them follows. For each, we verified our solution utilizing an "attacker" machine acting as a MITM using sslsplit [38], and a target "victim" machine running TrustBase. For some scenarios, the victim machine was implanted with our own CA in the distribution's shipped trust store or the store of a local user or application. Applications tested utilize the tools and libraries mentioned in section 6.2.

- **Hacked or coerced certificate authorities:** Attackers who have received a valid certificate through coercion, deception, or compromise of CAs are able to subvert even proper CA validation. Under TrustBase, administrators can choose to deploy pinning or notary plugins, which can detect the mismatch between the original and forged certificate, preventing the attacker from initiating a connection. We have developed plugins that perform these actions and verified that they prevent such attacks.

- **Local malicious root:** Attackers utilizing certificates that have been installed into an application or user trusted store will be trusted by many target applications. Even Google Chrome will ignore certificate pins in the presence of a certificate that links back to a locally-installed root certificate. TrustBase can protect against this by utilizing similar plugins to the preceding scenario.

- **Absence of status checking:** Many applications still do not check OCSP or Certificate Revocation Lists to determine if a received certificate is valid [29]. In these cases, attackers utilizing stolen certificates that have been reported can still perform a MITM. Administrators who want to prevent this from happening can add an OCSP or CRL plugin to the policy engine and ensure these checks for all applications on the machine. We have developed both OCSP and CRLSet plugins and verified that they perform status checks where applicable. For example, the OSCP plugin can be used to check certificates received by the Chrome browser, which does not do this natively.

- **Failure to validate hostnames:** Some applications properly validate signatures from a certificate back to a trusted root but do not verify that the hostname matches the one contained in the leaf certificate. This allows attackers to utilize any valid certificate, including those for hosts they legitimately control, to intercept traffic [20]. The TrustBase policy engine strictly validates the common name and all alternate names in a valid certificate against the intended hostname of the target host to eliminate this problem.

- **Lack of validation:** For applications that blindly accept all certificates, attackers need only send a self-signed certificate they generate on their own, or any other for which they have the private key, to MITM a connection. TrustBase prohibits this by default, as the policy engine ensures the certificate has a proper chain of signatures back to a trust anchor on the machine and performs the hostname validation described previously.

- **STARTTLS downgrade attack:** Opportunistic TLS begins with a plaintext connection. A downgrade attack occurs when an active attacker suppresses STARTTLS-related messages, tricking the endpoints into thinking one or the other does not

support STARTTLS. The net result is a continuation of the plaintext connection and possible sending of sensitive data (e.g., email) in the clear. TrustBase mitigates this attack by an option to enforce STARTTLS use. When STARTTLS is used to communicate with a given service, TrustBase records the host information. Future connections to that host are then required to upgrade via STARTTLS. If the host omits STARTTLS and prohibits its use, the connection is severed by TrustBase to prevent leaking sensitive information to a potential attacker.[6] TrustBase also allows the system administrator to configure a strict TLS policy, which disallows plaintext connections even if it has no prior data about whether a remote host supports STARTTLS.

5.4 Hardening

The following design principles strengthen the security of a TrustBase implementation. First, the traffic interceptor and handler components run in kernel space. Their small code size and limited functionality—handlers are simple finite state machines—make it more likely that formal methods and source code auditing will provide greater assurance that an implementation is correct. Second, the policy engine and plugins run in user space. This is where error-prone tasks such as certificate parsing and validation occur. The use of privilege separation [36] and sandboxing [30] techniques can limit the potential harm when any of these components is compromised. Third, plugins can only be installed and configured by an administrator, which prohibits unprivileged adversaries and malware from installing malicious authentication services. Finally, communications between the handlers, policy engine, and plugins are authenticated to prevent local malware from spoofing a certificate validation result.

TrustBase is designed to prevent a local, nonprivileged user from inadvertently or intentionally compromising the system. (1) Only privileged users can insert and remove the TrustBase kernel module, prohibiting an attacker from simply removing the module to bypass it. The same is true for plugins. (2) The communication between the kernel module component of TrustBase and the user space policy engine is performed via a custom Generic Netlink protocol that protects against nonprivileged users sending messages to the kernel. The protocol definition takes advantage of the Generic Netlink flag `GENL_ADMIN_PERM`, which enforces that selected operations associated with the custom protocol can only be invoked by processes that have administrative privileges for networking (the capability mapped to `CAP_NET_ADMIN` in Linux systems). This prevents a local attacker from using a local Netlink-utilizing process to masquerade as the policy engine to the kernel. (3) The policy engine runs as a nonroot, `CAP_NET_ADMIN` process that can be invoked only by a privileged user. (4) The configuration files, plugin directories, and binaries for TrustBase are write-protected to prevent unauthorized modifications from nonprivileged users. This protects against weakening of the configuration, disabling of plugins, shutting down or replacing the policy engine, or enabling of bogus plugins.

TrustBase stops traffic interception for a given flow as soon as it is identified as a non-TLS connection. Experimental results show that TrustBase has negligible overhead with respect to memory and time while tracking connections. Thus it is unlikely that an attacker could perform a denial-of-service attack on the machine by creating multiple network connections, TLS or otherwise, any easier than in the non-TrustBase scenario. Such an attack is more closely associated with firewall policies.

An attacker may seek to compromise TrustBase by crafting an artificial TLS handshake that results in some type of TrustBase failure, hoping to cause some kind of application error or termination. We reduce this attack surface by performing no parsing in the kernel except for TLS handshake records, which involves just the message type, length, and version headers. ASN.1 and other data sent to the policy engine are evaluated and parsed by standard openssl functions, which have undergone widespread scrutiny and use for many years. The TrustBase code has been made publicly available, and we invite others to audit the code. We note that, in the absence of the local proxy, TrustBase will not coerce an application to accept a certificate that the application would normally reject.

6 Evaluation

We evaluated the prototype of TrustBase to measure its performance, ensure compatibility with applications, and test its utility for deploying authentication services that can harden certificate validation.

6.1 Performance

To measure the overhead incurred by TrustBase, we instrumented our implementation to record the time required to establish a TCP connection, establish a TLS connection, and transfer a file of varying size (2MB - 500 GB). We tested TrustBase with two plugins, CA Validation and Certificate Pinning (see Section 6.5). The target host for these connections was a computer on the same local network as the client machine, to reduce the effect of latency and network noise. The host presented a valid certificate

[6] This could be further strengthened by checking DANE records to determine if the server supports STARTTLS. We are likewise interested in pursuing whether this technique can be used to protect against other types of downgrade attacks.

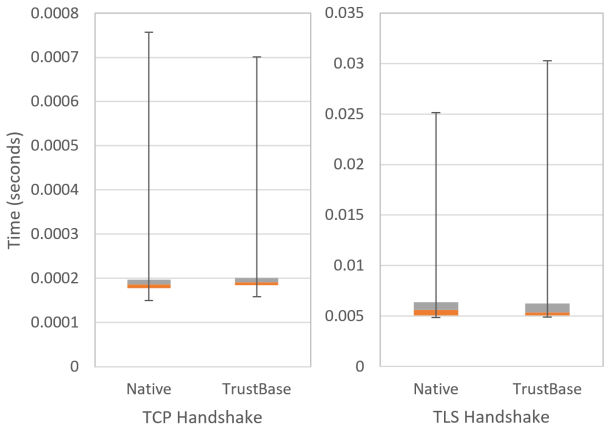

Figure 4: Handshake Timings for TCP (left) and TLS (right) handshakes with and without TrustBase running.

chain that also employed an intermediate authority, representing a realistic circumstance for web browsing and forcing plugins to execute all of their validity checks. Our testing used a modern PC running Fedora 21 and averaged across 1,000 trials.

Figure 4 shows boxplots that characterize the timing of TCP and TLS handshakes, with and without TrustBase active. There is no discernible difference for TCP handshake timings and the average difference is less than 10 microseconds, with neither configuration consistently beating the other in subsequent experiments. This is expected behavior because the traffic interceptor is extremely light-weight for TCP connections. Average TLS handshake times with and without TrustBase also have no discernible difference, with average handshake times for this experiment of 5.9 ms and 6.0 ms, respectively. Successive experiments showed again that neither average consistently beat the other. This means that the inherent fluctuations in system and network conditions account for more time than the additional control paths TrustBase introduces. This is also expected, as the brevity of TLS handling code, its place in the kernel, the use of efficient Netlink transport and other design choices were made with performance in mind.

Our experimentation with varying file sizes also exhibited no difference between native and TrustBase cases. Note that the TrustBase timings for the TLS handshake may increase if a particular plugin is installed that requires more processing time or relies on Internet queries to function, and that this overhead is inherent to that service and not the TrustBase core.

The memory footprint in our Linux prototype is also negligible. For each network connection, TrustBase temporarily stores less than 300 bytes of data, plus the length of any TLS handshake messages encountered. Connections not using TLS use even less memory than this and carry a zero-byte memory overhead once their nature has been determined and TrustBase ceases to monitor them.

6.2 Compatibility

One goal of TrustBase is to strengthen certificate authentication for existing, unmodified applications and to provide additional authentication services that strengthen the CA system. To meet this goal, TrustBase must be able to enforce proper authentication behavior by applications, as defined by the system administrator's configuration.

There are three possible cases for the policy engine to consider. (1) If a certificate has been deemed valid by both TrustBase and the application, the policy engine allows the original certificate data to be forwarded on to the application, where it is accepted naturally. (2) In the case where the application wishes to block a connection, regardless of the decision by TrustBase, the policy engine allows this to occur, since the application may have a valid reason to do so. We discuss in Section 3.5, the special case when a new authentication service is deployed that wishes to accept a certificate that the CA system normally would not. (3) In the case where validation with TrustBase fails, but the application would have allowed the connection to proceed, the policy engine blocks the connection by forwarding an intentionally invalid certificate to the application, which triggers any SSL application validation errors an application supports, and then subsequently terminates the connection.

We tested TrustBase with 34 popular applications and libraries and tools, shown in Table 1.[7] TrustBase successfully intercepted and validated certificates for all of them. For each library tested, and where applicable, we created sample applications that performed no validation and improper validation (bad checking of signatures, hostnames, and validity dates). We then verified that TrustBase correctly forced these applications to reject false certificates despite those vulnerabilities in each case. In addition, we observed that TrustBase caused no adverse behavior, such as timeouts, crashes, or unexpected errors.

6.3 Android Prototype

To verify that the TrustBase approach works on mobile platforms and is compatible with mobile applications, we built a prototype for Android. Source code can be found at `owntrust.org`.

Our Android implementation uses the `VPNService` so that it can be installed on an unaltered OS and without root permissions. The drawback of this choice is that only one VPN service can be active on the Android at a time. In the long-term, adding socket-level interception to the Android kernel would be the right architectural choice,

[7] These are a superset of the tools and libraries tested with CertShim

Library	Tool
C++	gnutls-cli
libcurl	curl
libgnutls	sslscan
libssl	openssl s_client
libnss	openssl s_time
JAVA	lynx
SSLSocketFactory	fetchmail
PERL	firefox
socket::ssl	chrome/chromium
PHP	mpop
fsockopen	w3m
php_curl	ncat
PYTHON	wget
httplib	steam
httplib2	thunderbird
pycurl	kmail
pyOpenSSL	pidgin
python ssl	
urllib, urllib2, urllib3	
requests	

Table 1: Common Linux libraries and tools compatible with TrustBase

and then TrustBase could use similar traffic interception techniques as with the Linux implementation.

The primary engineering consequence of using the `VPNService` on Android is that TrustBase must intercept IP packets from applications but emit TCP (or UDP) packets to the network. If it could use raw sockets, then TrustBase could merely transfer IP packets between the `VPNService` and the remote server. Unfortunately, the lowest level socket endpoint an Android developer can create is the Java `Socket` or `DatagramSocket`, which encapsulate TCP and UDP payloads respectively. Therefore, we must emulate IP, UDP and TCP to facilitate communication between the `VPNService` and the sockets used to communicate with remote hosts. For TCP, this involves maintaining connection state, emulating reliability, and setting appropriate flags (SYN, ACK, etc.) for TCP traffic.

To verify compatibility with mobile applications, we tested 16 of the most popular Android applications: Chrome, YouTube, Pandora, Gmail, Pinterest, Instagram, Facebook, Google Play Store, Twitter, Snapchat, Amazon Shopping, Kik, Netflix, Google Photos, Opera, and Dolphin. TrustBase on Android successfully intercepted and strengthened certificate validation for all of them.

6.4 Windows Prototype

To demonstrate that the TrustBase approach works on Windows, we also built a prototype for Windows 10. Source code can be found at `owntrust.org`.

The traffic interceptor component of TrustBase on Windows is implemented utilizing the native Windows Filtering Platform (WFP) API, acting as a kernel-mode driver. Reliance on the WFP reduced the code necessary to provide traffic interception capabilities and also made them easy to maintain, given that the Windows kernel code is not open source. As on Linux, this kernel code is event-driven, collects connection information, and transmits it to a userspace policy engine for processing and decision making. The policy engine is patterned after its Linux counterpart, supports both Python and C plugins, and uses native Windows libraries where possible (e.g., Microsoft's CryptoAPI and native threading APIs).

The nature of the Windows root certificate store allows TrustBase to avoid utilizing a TLS Proxy in cases where overriding the CA system is desired (see Section 3.5). Windows has the ability to dynamically alter the root certificate store during runtime, and applications using the CA system will be immediately subject to those changes. This allows TrustBase to dynamically add self-signed certificates to the root store when the policy engine deems them trustworthy. Through this mechanism TrustBase can override the CA system by placing a validated certificate in the root store before the application obtains and validates it against the root store. TrustBase maintains identifying hashes of all the certificates added to the root store and removes them when the connections using them are terminated. As on Linux, applications that use their own private certificate stores cannot have their validation rejections overridden using this methodology.

6.5 Utility

To validate the utility of TrustBase, we implemented eight useful authentication services. Table 2 describes each of these services. These services illustrate the types of control that TrustBase can provide to an administrator in securing TLS on a system. The CA validation plugin ensures that all applications on the system perform appropriate checks when validating certificates received through TLS (hostname, basic constraints, expiration, etc.). The whitelist represents a more manual, customized approach to validation, likely to be used in conjunction with other services to handle edge cases. Our certificate pinning and certificate revocation services enforce more advanced checks that are usually reserved for individual applications but can now be deployed system-wide. Note that this includes the deployment of Google's CRLSets checks, which are normally reserved for Chromium browsers only. This addresses the limitation noted by [11] concerning the isolation of newer validation technologies in browser code. The Notary and DANE services can be leveraged to trust additional channels of information aside from CA signatures and revocation lists. Finally, our cipher suite auditor service allows system administrators to pre-

CA Validation	Enforces standard certificate validation using `openssl` functions and standard practices for validating hostnames, Basic Constraints, dates, etc.
Whitelist	Stores a set of certificates that are always considered valid for their respective hosts, such as self-signed certificates.
Certificate Pinning	Uses Trust On First Use to pin certificates for any host; expired certificates are replaced by the next certificate received by a connection to that domain.
Certificate Revocation	Checks OCSP to determine whether the certificate has been revoked.
CRLSet Blocking	Checks Google's CRLSet to determine whether the certificate has been blocked, extending Chrome's protection to all apps.
DANE	Uses the DNS system to distribute public keys in a TLSA record [21].
Notary	Based on ideas presented by Perspectives [41] and Convergence [31], it connects securely to one or more notary servers to validate the certificate received by the client is the same one that is seen by the notaries.
Cipher Suite Auditor	Uses Client Hello and Server Hello information, along with a configuration with secure defaults, to disallow weak cipher suites. It can also require that certain TLS extensions be employed (e.g., TACK[32]).

Table 2: Authentication and Security services implemented with TrustBase

vent connections that attempt to utilize weak cipher suites and signing algorithms, using the additional handshake information provided to all plugins.

7 Future Work

TrustBase explores the benefits and drawbacks of providing authentication as an operating system service and giving administrators control over how all authentication decisions on their machines are made. In doing so, a step has been taken toward empowering administrators to control secure connections on their machines. However, some drawbacks have been noted, such as the reliance on a local proxy to support TLS 1.3 interception and CA overriding in some cases on Linux. These issues are caused by applications dictating the security of the machine's connections, using their own (or third party) security features and keys, reducing operating system and administrator control.

We are currently investigating further steps into this territory to provide great administrator control of security without some of these drawbacks. One such step is providing TLS as an operating system service, meaning that the operating system provides encryption for applications, not just authentication. Current TLS libraries are a burden on application developers, who are often not security experts. In addition, developers do not necessarily share the same security goals as the vendors or administrators who configure the systems upon which applications run. By providing TLS as an operating system service, application developers are relieved of this burden and the OS can invoke the TrustBase validation API natively. This removes the need for developers to explicitly invoke the validation API, and provides the OS with visibility and control over all TLS data, including TLS 1.3 handshakes, as the OS becomes the de facto TLS client. Such a measure enables system-wide deployment of security measures, such as cipher suite customization, TLS extension deployment, and responses to CVEs. This also allows OS vendors and system administrators an easier upgrade path for TLS versions.

Since network application developers are already familiar with the POSIX socket API, we are working on providing TLS as a protocol type in the socket API, the same way the OS provides TCP and UDP protocols as a service. In contrast to using a userspace library, this approach allows network application developers unfamiliar with security to operate in a well-known environment, utilizes an existing OS API that can be shared by many different platform implementations, and allows strict configuration and control by administrators. By creating a socket using a new `IPPROTO_TLS` parameter (as opposed to `IPPROTO_TCP`), developers can use the `bind`, `connect`, `send`, `recv`, and other socket API calls with which thy are already familiar, focusing solely on application data and letting the OS handle all TLS functionality. The generalized `setsockopt` and `getsockopt` are available to specify remote hostnames and additional options to the OS TLS service without violating the existing socket API.

8 Conclusion

We have explored how to fix broken authentication in existing applications, while also providing a platform for improved authentication services. To solve these problems we used two guiding principles—centralizing authentication as an operating system service and giving system administrators control over authentication policy. Following these two principles, we designed the TrustBase architecture for certificate authentication, meeting our design goals of securing existing applications, strengthening the CA system, providing full application coverage, enabling universal deployment, and imposing negligible overhead. We have presented a research prototype for TrustBase on Linux, discussed how we hardened this implementation, provided a security analysis, and evaluated its performance. We have provided source code for Linux, Android, and Windows prototypes. Finally, we have written eight authentication services to demonstrate the utility of this approach, extending CA hardening to all applications.

9 Acknowledgments

The authors thank the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1528022 and research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) via contract number HHSP233201600046C. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security. Also, this work was supported by Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] ALICHERRY, M., AND KEROMYTIS, A. D. Doublecheck: Multi-path verification against man-in-the-middle attacks. In *Symposium on Computers and Communications (ISCC)* (2009), IEEE, pp. 557–563.

[2] AMANN, B., VALLENTIN, M., HALL, S., AND SOMMER, R. Extracting certificates from live traffic: A near real-time SSL notary service. Tech. rep., TR-12-014, ICSI Nov. 2012, 2012.

[3] AMANN, B., VALLENTIN, M., HALL, S., AND SOMMER, R. Revisiting SSL: A large-scale study of the internet's most trusted protocol. Tech. rep., TR-12-015, ICSI Dec. 2012, 2012.

[4] BATES, A., PLETCHER, J., NICHOLS, T., HOLLEMBAEK, B., TIAN, D., BUTLER, K. R., AND ALKHELAIFI, A. Securing SSL certificate verification through dynamic linking. In *Conference on Computer and Communications Security (CCS)* (2014), ACM.

[5] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Symposium on Security and Privacy (SP)* (2014), IEEE.

[6] CONTI, M., DRAGONI, N., AND GOTTARDO, S. MITHYS: Mind the hand you shake-protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*. Springer, 2013, pp. 65–81.

[7] DE CARNAVALET, X. D. C., AND MANNAN, M. Killed by proxy: Analyzing client-end TLS interception software. In *Network and Distributed System Security Symposium (NDSS)* (2016), Internet Society.

[8] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[9] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., KASTEN, J., BURSZTEIN, E., LIDZBORSKI, N., THOMAS, K., ERANTI, V., BAILEY, M., AND HALDERMAN, J. A. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *Internet Measurement Conference (IMC)* (2015), ACM.

[10] DURUMERIC, Z., KASTEN, J., BAILEY, M., AND HALDERMAN, J. A. Analysis of the HTTPS certificate ecosystem. In *Internet Measurement Conference (IMC)* (2013), ACM.

[11] DURUMERIC, Z., MA, Z., SPRINGALL, D., BARNES, R., SULLIVAN, N., BURSZTEIN, E., BAILEY, M., HALDERMAN, J. A., AND PAXSON, V. The security impact of HTTPS interception. In *Network and Distributed System Security Symposium (NDSS)* (2017), Internet Society.

[12] ECKERSLEY, P., AND BURNS, J. An observatory for the SSLiverse. `http://www.eff.org/files/DefconSSLiverse.pdf`, 2010.

[13] ECKERSLEY, P., AND BURNS, J. The (decentralized) SSL observatory. In *USENIX Security Symposium* (2011).

[14] (EFF), E. F. F. The Sovereign Keys Project. `http://www.eff.org/sovereign-keys/`, 2011.

[15] ENGERT, K. MECAI - mutually endorsing CA infrastructure. `http://kuix.de/mecai`. Accessed: March 2013.

[16] EVANS, C., AND PALMER, C. Certificate pinning extension for HSTS. `http://tools.ietf.org/html/draft-evans-palmer-hsts-pinning-00`. Accessed: 22 March, 2013.

[17] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Conference on Computer and Communications Security (CCS)* (2012), ACM.

[18] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL development in an appified world. In *Conference on Computer Communications Security (CCS)* (2013), ACM.

[19] FOSTER, I. D., LARSON, J., MASICH, M., SNOEREN, A. C., SAVAGE, S., AND LEVCHENKO, K. Security by any other name: On the effectiveness of provider based email security. In *Conference on Computer and Communications Security (CCS)* (2015), ACM.

[20] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Conference on Computer and Communications Security (CCS)* (2012), ACM.

[21] HOFFMAN, P., AND SCHLYTER, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA, RFC 6698. https://datatracker.ietf.org/doc/rfc6698, 2012. Accessed: 24 Feb, 2014.

[22] HOLZ, R., AMANN, J., MEHANI, O., WACHS, M., AND KAAFAR, M. A. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium (NDSS)* (2016), Internet Society.

[23] HOLZ, R., RIEDMAIER, T., KAMMENHUBER, N., AND CARLE, G. X.509 forensics: Detecting and localising the SSL/TLS men-in-the-middle. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2012, pp. 217–234.

[24] HUANG, L.-S., RICE, A., ELLINGSEN, E., AND JACKSON, C. Analyzing forged SSL certificates in the wild. In *Symposium on Security and Privacy (SP)* (2014), IEEE.

[25] KEIZER, G. Hackers spied on 300,000 Iranians using fake Google certificate. http://www.computerworld.com/article/2510951/cybercrime-hacking/hackers-spied-on-300-000-iranians-using-fake-google-certificate.html. Accessed: 27 October, 2015.

[26] KIM, T. H.-J., HUANG, L.-S., PERRING, A., JACKSON, C., AND GLIGOR, V. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *International Conference on World Wide Web (WWW)* (2013).

[27] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency, IETF RFC 6962. http://tools.ietf.org/html/rfc6962, Jun 2013.

[28] LIGH, M. H., CASE, A., LEVY, J., AND WALTERS, A. *The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons, 2014.

[29] LIU, Y., TOME, W., ZHANG, L., CHOFFNES, D., LEVIN, D., MAGGS, B., MISLOVE, A., SCHULMAN, A., AND WILSON, C. An end-to-end measurement of certificate revocation in the web's PKI. In *Internet Measurement Conference (IMC)* (2015), ACM.

[30] MAASS, M., SALES, A., CHUNG, B., AND SUNSHINE, J. A systematic analysis of the science of sandboxing. *PeerJ Computer Science 2* (2016), e43.

[31] MARLINSPIKE, M. SSL and the future of authenticity. *Black Hat USA* (2011).

[32] MARLINSPIKE, M., AND PERRIN, T. Trust assertions for certificate keys. http://tack.io/, 2013.

[33] NYGREN, E. Reaching toward universal TLS SNI. https://blogs.akamai.com/2017/03/reaching-toward-universal-tls-sni.html. Accessed: 21 June, 2017.

[34] O'NEILL, M., RUOTI, S., SEAMONS, K., AND ZAPPALA, D. TLS proxies: Friend or foe? In *Internet Measurement Conference (IMC)* (2016), ACM.

[35] ONWUZURIKE, L., AND DE CRISTOFARO, E. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)* (2015), ACM.

[36] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *USENIX Security* (2003), vol. 3.

[37] RISTIĆ, I. Bulletproof SSL and TLS. *Feisty Duck* (2014).

[38] ROETHLISBERGER, D. Sslsplit. https://www.roe.ch/SSLsplit. Accessed: 11 July, 2015.

[39] RYAN, M. D. Enhanced certificate transparency and end-to-end encrypted mail. In *Network and Distributed System Security Symposium (NDSS)* (2014), Internet Society.

[40] SOGHOIAN, C., AND STAMM, S. Certified lies: Detecting and defeating government interception attacks against SSL (short paper). In *Financial Cryptography and Data Security*. Springer, 2012, pp. 250–259.

[41] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference* (2008).