The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software

Martin Georgiev
The University of Texas at Austin

Subodh Iyengar
Stanford University

Suman Jana
The University of Texas at Austin

Rishita Anubhai
Stanford University

Dan Boneh
Stanford University

Vitaly Shmatikov
The University of Texas at Austin

ABSTRACT

SSL (Secure Sockets Layer) is the de facto standard for secure Internet communications. Security of SSL connections against an active network attacker depends on correctly validating public-key certificates presented when the connection is established.

We demonstrate that SSL certificate validation is completely broken in many security-critical applications and libraries. Vulnerable software includes Amazon's EC2 Java library and all cloud clients based on it; Amazon's and PayPal's merchant SDKs responsible for transmitting payment details from e-commerce sites to payment gateways; integrated shopping carts such as osCommerce, ZenCart, Ubercart, and PrestaShop; AdMob code used by mobile websites; Chase mobile banking and several other Android apps and libraries; Java Web-services middleware—including Apache Axis, Axis 2, Codehaus XFire, and Pusher library for Android—and *all* applications employing this middleware. Any SSL connection from any of these programs is insecure against a man-in-the-middle attack.

The root causes of these vulnerabilities are badly designed APIs of SSL implementations (such as JSSE, OpenSSL, and GnuTLS) and data-transport libraries (such as cURL) which present developers with a confusing array of settings and options. We analyze perils and pitfalls of SSL certificate validation in software based on these APIs and present our recommendations.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

Keywords

SSL, TLS, HTTPS, public-key infrastructure, public-key certificates, security vulnerabilities

1. INTRODUCTION

Originally deployed in Web browsers, SSL (Secure Sockets Layer) has become the de facto standard for secure Internet communications. The main purpose of SSL is to provide end-to-end security against an active, man-in-the-middle attacker. Even if the network is completely compromised—DNS is poisoned, access points and routers are controlled by the adversary, etc.—SSL is intended to guarantee confidentiality, authenticity, and integrity for communications between the client and the server.

Authenticating the server is a critical part of SSL connection establishment.[1] This authentication takes place during the SSL handshake, when the server presents its public-key certificate. In order for the SSL connection to be secure, the client must carefully verify that the certificate has been issued by a valid certificate authority, has not expired (or been revoked), the name(s) listed in the certificate match(es) the name of the domain that the client is connecting to, and perform several other checks [14, 15].

SSL implementations in Web browsers are constantly evolving through "penetrate-and-patch" testing, and many SSL-related vulnerabilities in browsers have been repaired over the years. SSL, however, is also widely used in *non-browser software* whenever secure Internet connections are needed. For example, SSL is used for (1) remotely administering cloud-based virtual infrastructure and sending local data to cloud-based storage, (2) transmitting customers' payment details from e-commerce servers to payment processors such as PayPal and Amazon, (3) logging instant messenger clients into online services, and (4) authenticating servers to mobile applications on Android and iOS.

These programs usually do not implement SSL themselves. Instead, they rely on SSL libraries such as OpenSSL, GnuTLS, JSSE, CryptoAPI, etc., as well as higher-level data-transport libraries, such as cURL, Apache HttpClient, and *urllib*, that act as wrappers around SSL libraries. In software based on Web services, there is an additional layer of abstraction introduced by Web-services middleware such as Apache Axis, Axis 2, or Codehaus XFire.

Our contributions. We present an in-depth study of SSL connection authentication in non-browser software, focusing on how diverse applications and libraries on Linux, Windows, Android, and iOS validate SSL server certificates. We use both white- and black-box techniques to discover vulnerabilities in validation logic. Our main conclusion is that *SSL certificate validation is completely broken in many critical software applications and libraries*. When presented with self-signed and third-party certificates—including a certificate issued by a legitimate authority to a domain called `AllYourSSLAreBelongTo.us` —they establish SSL connections and send their secrets to a man-in-the-middle attacker.

[1]SSL also supports client authentication, but we do not analyze it in this paper.

This is exactly the attack that SSL is intended to protect against. It does not involve compromised or malicious certificate authorities, nor forged certificates, nor compromised private keys of legitimate servers. The only class of vulnerabilities we exploit are logic errors in client-side SSL certificate validation.

The root cause of most of these vulnerabilities is the terrible design of the APIs to the underlying SSL libraries. Instead of expressing high-level security properties of network tunnels such as confidentiality and authentication, these APIs expose low-level details of the SSL protocol to application developers. As a consequence, developers often use SSL APIs incorrectly, misinterpreting and misunderstanding their manifold parameters, options, side effects, and return values. In several cases, we observed developers introducing new vulnerabilities when attempting to "fix" certificate validation bugs. Furthermore, deveopers often do not understand which security properties are or are not provided by a given SSL implementation: for example, they use SSL libraries that do not validate certificates even when security is essential (e.g., connecting to a payment processor). More prosaic, yet deadly causes include intermediate layers of the software stack silently disabling certificate validation and developers turning off certificate validation accidentally (e.g., for testing) or intentionally.

2. OVERVIEW OF OUR RESULTS

Our study uncovered a wide variety of SSL certificate validation bugs. Affected programs include those responsible for managing cloud-based storage and computation, such as Amazon's EC2 Java client library and Elastic Load Balancing API Tools, Apache Libcloud, Rackspace iOS client, and Windows-based cloud storage clients such as ElephantDrive and FilesAnywhere.

Java-based Web-services middleware, such as Apache Axis, Axis 2, and Codehaus XFire, is broken, too. So is the Android library for Pusher notification API and Apache ActiveMQ implementation of Java Message Service. All programs employing this middleware are *generically* insecure.

Certificate validation bugs are pervasive in "merchant SDKs," which typically run on e-commerce servers (e.g., online stores) and are responsible for transmitting customers' financial details to payment processing gateways. Broken libraries include Amazon Flexible Payments Service (both Java and PHP), as well as PayPal Payments Standard and PayPal Invoicing (both in PHP), PayPal Payments Pro, Mass Pay, and Transactional Information SOAP (all in Java). Most payment modules for integrated shopping carts, such as ZenCart, Ubercart, PrestaShop, and osCommerce, do not validate certificates, either. A man-in-the-middle attack enables the attacker to harvest credit card numbers, names, addresses, etc. of the customers of any merchant who uses one of these programs for payment processing. Mobile app providers who use AdMob's sample code to link app instances to their AdMob accounts are vulnerable, too, enabling the attacker to capture the developer's account credentials and gain access to all of her Google services.

Instant messenger clients such as Trillian and AIM do not validate certificates correctly, either. A man-in-the-middle attack on Trillian yields login credentials for all Google (including Gmail), Yahoo!, and Windows Live services (including SkyDrive).

Not the most interesting technically, but perhaps the most devastating (because of the ease of exploitation) bug is the broken certificate validation in the Chase mobile banking app on Android. Even a primitive network attacker—for example, someone in control of a malicious Wi-Fi access point—can exploit this vulnerability to harvest the login credentials of Chase mobile banking customers. Other insecure Android software includes Breezy, a "secure" printing app, and the ACRA library for application crash reporting.

In summary, **SSL connections established by any of the above programs are insecure against a man-in-the-middle attack.** All vulnerabilities have been empirically confirmed.

Causes. For the most part, the actual SSL libraries used in these programs are correct. Yet, regardless of which well-known library the software relies on—whether JSSE, OpenSSL, GnuTLS, or CryptoAPI, used directly or wrapped into a data-transport library such as Apache HttpClient or cURL—it often finds a way to end up with broken or disabled SSL certificate validation.

The primary cause of these vulnerabilities is the developers' misunderstanding of the numerous options, parameters, and return values of SSL libraries. For example, Amazon's Flexible Payments Service PHP library attempts to enable hostname verification by setting cURL's `CURLOPT_SSL_VERIFYHOST` parameter to `true`. Unfortunately, the correct, default value of this parameter is 2; setting it to `true` silently changes it to 1 and disables certificate validation. PayPal Payments Standard PHP library *introduced* the same bug when updating a previous, broken implementation. Another example is Lynx, a text-based browser which is often used programmatically and thus included in our study. It checks for self-signed certificates—but only if GnuTLS's certificate validation function returns a negative value. Unfortunately, this function returns 0 for certain errors, including certificates signed by an untrusted authority. Chain-of-trust verification in Lynx is thus broken.

Developers often misunderstand security guarantees provided by SSL libraries. For example, JSSE (Java Secure Socket Extension) has multiple interfaces for managing SSL connections. The "advanced" `SSLSocketFactory` API silently skips hostname verification if the algorithm field in the SSL client is NULL or an empty string rather than HTTPS. This is mentioned in passing in the JSSE reference guide, yet many Java implementations of SSL-based protocols use `SSLSocketFactory` without performing their own hostname verification. Vulnerable libraries include Apache HttpClient version 3.* and the Weberknecht implementation of WebSockets. *Any* Java program based on these libraries is generically insecure against a man-in-the-middle attack. Vulnerable programs include SOAP Web-services middleware such as Apache Axis and Codehaus XFire, as well as any software built on top of it (for example, Amazon's EC2 client library), any Android app that uses Pusher API to manage real-time messaging (for example, GitHub's Gaug.es), clients of Apache ActiveMQ servers, etc.

Other bugs include using incorrect regular expressions for hostname matching, not checking the results of certificate validation correctly, accidentally or deliberately disabling validation, etc.

Lessons. First, the state of adversarial testing appears to be exceptionally poor even for critical software such as mobile banking apps and merchant SDKs responsible for managing secure connections to payment processors. Most of the vulnerabilities we found should have been discovered during development with proper unit testing.

Second, many SSL libraries are unsafe by default, requiring higher-level software to correctly set their options, provide hostname verification functions, and interpret return values. As we show, software that relies on these libraries is often not up to the task.

Third, even safe-by-default libraries, such as cURL's wrapper around OpenSSL, are misused by developers that misinterpret the meaning of various options. This calls for better documentation and more rigorous formalization of API semantics. In particular, APIs should present high-level abstractions to developers, such as "confidential and authenticated tunnel," as opposed to requiring them to explicitly deal with low-level details such as hostname verification.

Fourth, SSL bugs are often hidden deep inside layers of middleware, above the actual SSL implementation but below the application, making the problem hard to locate and repair, and effectively taking it out of application developers' hands.

Fifth, least interesting technically but still critically important, we observed many cases of developers deliberately disabling certificate validation, while assuring both users and higher-level programs that SSL is being supported but not informing them that protection against active attacks has been turned off.

3. OVERVIEW OF SSL

3.1 Threat model

We assume an *active, man-in-the-middle network attacker* who may control network routers or switches, Wi-Fi access points, and/or DNS. She may also control one or more servers and possess valid SSL certificates for these servers. When an SSL client attempts to connect to a legitimate server, the attacker can mislead it about the server's network address (e.g., through DNS poisoning) and trick it into connecting to an attacker-controlled server instead.

Our attacker (1) does not have access to the private keys of legitimate servers, (2) does not control any certificate authorities, (3) cannot forge certificates. Even if she succeeds in spoofing the address of a legitimate server, a correctly implemented SSL client should refuse to accept the malicious server's certificate because of a mismatch between the name(s) on the certificate and the domain to which the client is connecting.

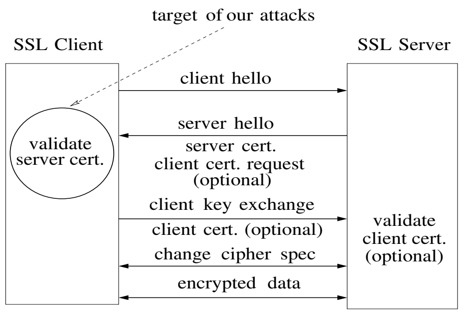

Figure 1: Simplified overview of SSL handshake.

3.2 SSL certificate validation

An SSL connection starts with a handshake between the client and the server. The handshake protocol is summarized in Figure 1; see RFC 6101 [16] for a complete description.

We focus on the client's validation of the server certificate. All SSL implementations we tested use X.509 certificates. The complete algorithm for validating X.509 certificates can be found in RFC 5280 [15] and RFC 2818 [14]. In this paper, we consider two of the checks; both are critical for security against active attacks.

Chain-of-trust verification. Each X.509 certificate has an "issuer" field that contains the name of the certificate authority (CA) that issued the certificate. Every SSL client is configured with a list of certificates for trusted root CAs.

In addition to its own certificate, the server sends the certificate of the issuing CA. If the issuing CA is not a root CA, the server also sends a list of certificates of higher-level CAs all the way to a root CA. The client attempts to build a chain starting from the server's certificate at the bottom. Each certificate in the chain must be signed by the CA immediately above it; the root CA must be one of the client's trusted CAs. The client also verifies that the certificates have not expired and that the certificates of the intermediate CAs have the CA bit set in the "Basic Constraints" field.

Hostname verification. After the chain of trust is established, the client must verify the server's identity. RFC 2818 advises the implementors to use "SubjectAltNames" as the main source of server identifiers and support "Common Name" for backward compatibility only, but most of the software we tested does it the other way around and checks "Common Name" first. After building the list of server identifiers, the client attempts to match the fully qualified DNS name of the requested server to one of the identifiers.

If the client finds an exact match in the list of server identifiers, verification is done by straightforward string comparison. The client may also find a wildcard name in the list of identifiers. The rules for wildcard matching are fairly complex [14, 17], especially concerning international character sets.

Certificate revocation and X.509 extensions. This paper focuses on verifying the server's identity, but full certificate validation involves many more checks. These checks are essential for security, yet are handled poorly or not at all by non-browser software.

For example, some SSL libraries such as OpenSSL implement certificate revocation, but require the application to provide the certificate revocation list (CRL). The applications we analyzed do not avail themselves of this facility. Furthermore, libraries such as JSSE require the application to check validity of the CRL on its own. Most applications don't bother. Other SSL libraries, such as Python's *ssl*, do not expose a method for CRL checking.

Some X.509 certificate extensions contain security-critical information such as key usage (e.g., is the CA allowed to use this key for signing certificates?), name constraints (restricting the names that a sub-CA can certify), and certificate policies, described in RFC 2527 [13]. For instance, a CA may assign different levels of trust to different sub-CAs, but the application must provide a policy that takes advantage of this information. In practice, these extensions are largely neglected. For example, until recently OpenSSL did not validate name constraints correctly, while cURL does not even have an interface for specifying the application's certificate policy.

Attacks exploiting improper treatment of certificate revocation and X.509 extensions are somewhat different from the "pure" man-in-the-middle model considered in this paper. We leave their detailed analysis to future work.

4. SSL ABSTRACTIONS

Depending on its needs, an application can "plug" into SSL at different levels of abstraction. At the lowest level, there are many popular SSL implementations with different features, licenses, and hardware requirements: OpenSSL, JSSE, CryptoAPI, NSS, yaSSL, GnuTLS, BouncyCastle, and others. These libraries are mostly oblivious to protocols transported over SSL. Therefore, to avoid having to parse HTTP messages on their own, applications that involve HTTP over SSL (HTTPS) typically do not use them directly. Instead, they employ one of the many HTTPS libraries (see Section 4.2), which in turn use SSL libraries internally. Applications that use SOAP- or REST-based Web services require additional middleware on top of HTTPS or WebSockets (see Figure 2).

4.1 SSL libraries

OpenSSL. OpenSSL only provides chain-of-trust verification; applications must supply their own hostname verification code. This is typical for low-level SSL libraries. Different application-layer protocols such as HTTPS, LDAP, etc. have different notions of what constitutes a valid hostname and what it means for a hostname to match the name(s) listed in the certificate. Therefore, hostname

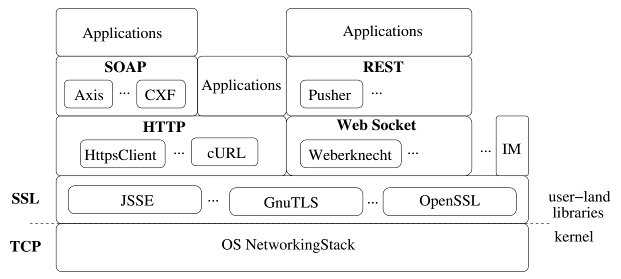

Figure 2: Protocol stack.

verification must be managed either by the application itself, or by a data-transport wrapper such as cURL.

Proper hostname verification for OpenSSL and CryptoAPI is discussed in [21, Chap. 10.8], assuming the chain of trust has been verified correctly. As discussed in [21, Chap. 10.5], the latter is error-prone due to the complexity of the underlying API. OpenSSL allows applications to customize chain-of-trust verification by providing a callback function or modifying configuration variables such as "verify depth" and "verify mode" as shown in Figure 3.

A program using OpenSSL can perform the SSL handshake by invoking the `SSL_connect` function. A high-level overview of the handling of different configurations and callbacks is shown in Algorithm 1. They can have complex interactions.

Some certificate validation errors are signaled through the return values of `SSL_connect`, while for other errors `SSL_connect` returns OK but sets internal "verify result" flags. Applications must call `SSL_get_ verify_result` function to check if any such errors occurred. This approach is error-prone (see Section 7.6).

Algorithm 1 Outline of SSL_connect control flow.

```
while chain of trust contains no trusted CA do
  if chain length <verify_depth then
    Try to extend chain of trust by 1 level
    Set ERROR appropriately if any error
  else
    Set ERROR to 'incomplete chain'
  end if
  if ERROR then
    verify_result = error
    if verify_callback == NULL then
      if verify_mode != 0 then
        Print error and terminate connection.
      end if
    else
      ret = verify_callback(preverify_ok = 0, . . . )
      if (verify_mode != 0) and (ret == 0) then
        Print error and terminate connection.
      end if
    end if
    if ERROR is not related to incorrect parsing then
      return 1
    else
      return ERROR
    end if
  else
    ret = verify_callback(preverify_ok = 1, . . . )
    if (verify_mode != 0) and (ret == 0) then
      Print error and terminate connection.
    end if
  end if
end while
return 1
```

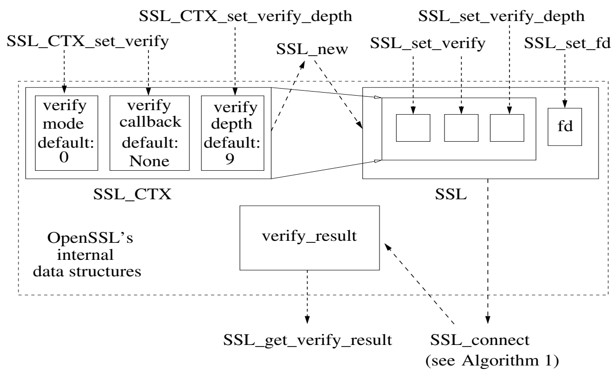

Figure 3: OpenSSL API for setting up SSL connections with the default chain-of-trust verification.

The certificate validation function in GnuTLS, `gnutls_certif- icate_verify_peers2`, has similarly atrocious error reporting. It takes a reference to `tls_status` as an argument and sets it to an appropriate error code if validation fails. For some errors (e.g., insufficient credentials or no certificate found), it returns a negative value; for others (e.g., self-signed certificate), it sets the error code but returns zero. In Section 7.4 we show that application developers misunderstand this complex relationship between the error status and the return value, resulting in broken certificate validation.

JSSE. Java Secure Socket Extension (JSSE) provides numerous interfaces through which Java applications—including Android mobile apps—can establish SSL connections.

The low-level API is `SSLSocketFactory`. Depending on how the SSL client is created, this API may or may not perform hostname verification. The following code sample is taken from `X509Trust ManagerImpl.checkIdentity` in Java 6 update 31:

```
private void checkIdentity(String hostname,
     X509Certificate cert, String algorithm)
   throws CertificateException {
   if (algorithm != null && algorithm.length() != 0) {
               ....
      if (algorithm.equalsIgnoreCase("HTTPS")) {
         HostnameChecker.getInstance(HostnameChecker.TYPE
              _TLS).match(hostname, cert);
      } else if (algorithm.equalsIgnoreCase("LDAP")) {
         HostnameChecker.getInstance(HostnameChecker.TYPE
              _LDAP).match(hostname, cert);
       } else {
         throw new CertificateException(
        "Unknown identification algorithm: " + algorithm);
     }
   }
}
```

The `checkIdentity` method throws an exception if the algorithm field is set to anything other than HTTPS or LDAP. This is different from, for example, OpenSSL, which returns a value even if verification fails and expects the application to check this value.

JSSE APIs such as `HttpsClient` and `HttpsURLConnection` call `try SetHostnameVerification` when creating SSL clients. This method sets the algorithm field to HTTPS. The above code thus invokes `HostnameChecker` and verifies the name in the certificate.

If the algorithm field in the client data structure is NULL or an empty string, `checkIdentity` silently skips hostname verification without throwing an exception. We conjecture that this behavior is designed to accommodate implementors of certificate-based protocols other than HTTPS or LDAP who may want to re-use

JSSE's default trust manager for chain-of-trust verification but provide their own, protocol-specific hostname verification.

On February 14, 2012, Java 7 update 3 was released. The code for certificate validation is different from Java 6, but its behavior is similar: if the algorithm field is NULL or an empty string, `checkIdentity` is never invoked.

```
private void checkTrusted(X509Certificate[] chain,
     String authType, Socket socket, boolean isClient)
     throws CertificateException {
    ...
    // check endpoint identity
    String identityAlg = sslSocket.getSSLParameters().
        getEndpointIdentificationAlgorithm();
    if (identityAlg != null && identityAlg.length != 0)
        {
       String hostname = session.getPeerHost();
       checkIdentity(hostname, chain[0], identityAlg);
    }
}
```

In SSL clients created using "raw" `SSLSocketFactory` (as opposed to `HttpsClient` or `HttpsURLConnection` wrappers), the algorithm field is NULL, thus JSSE does not perform hostname verification. The responsibility for hostname verification is delegated to the software running on top of JSSE. This feature is not explained in the API documentation. Instead, the following warning can be found deep inside the JSSE reference guide:[2]

> When using raw SSLSockets/SSLEngines you should always check the peer's credentials before sending any data. The SSLSocket and SSLEngine classes do not automatically verify that the hostname in a URL matches the hostname in the peer's credentials. An application could be exploited with URL spoofing if the hostname is not verified.

The prevalence of Java software that uses `SSLSocketFactory` to create SSL clients yet does *not* perform hostname verification (see Section 4.2) suggests that developers are not aware of this feature. The existence of alternative JSSE interfaces that *do* perform hostname verification only increases the confusion.

4.2 Data-transport libraries

In practice, most applications rely on data-transport frameworks to establish HTTPS connections. These frameworks use SSL libraries internally in a way that is usually opaque to applications.

Apache HttpClient. Apache HttpClient[3] is a client-side HTTP(S) Java library based on JDK. The latest version is 4.2.1, published on June 29, 2012, but most existing software employs older, 3.* versions. Apache HttpClient is used extensively in Web-services middleware such as Apache Axis 2 (see Section 8) because native JDK does not support SOAP Web services. Furthermore, Apache HttpClient provides better performance than JDK for functionalities such as sending HTTP POST requests.

Apache HttpClient uses JSSE's `SSLSocketFactory` to establish SSL connections. As explained in Section 4.1, this means that Apache HttpClient must perform its own hostname verification. This leads to numerous vulnerabilities in software based on older versions on HttpClient that do not verify hostnames (Section 7.5).

Furthermore, Apache HttpClient uses HttpHost data structure to describe HTTP(S) connections. HttpHost does not have any internal consistency checks: for example, it allows connections to port 443 to have HTTP as the scheme. In Section 7.8, we show how this leads to errors even in code implemented by SSL experts.

Weberknecht. Weberknecht[4] is a Java implementation of the WebSockets protocol. It uses `SSLSocketFactory` but does not perform its own hostname verification. Any Java program that employs Weberknecht is vulnerable to a man-in-the-middle attack.

cURL. cURL[5] is a popular tool and library (*libcurl*) for fetching data from remote servers. Since version 7.10, cURL validates SSL certificates by default. Internally, it uses OpenSSL to verify the chain of trust and verifies the hostname itself. This functionality is controlled by parameters `CURLOPT_SSL_VERIFYPEER` (default value: true) and `CURLOPT_SSL_VERIFYHOST` (default value: 2).

This interface is almost perversely bad. The `VERIFYPEER` parameter is a boolean, while a similar-looking `VERIFYHOST` parameter is an integer. The following quote from the cURL manual explains the meaning of `CURLOPT_SSL_VERIFYHOST`:

> 1 to check the existence of a common name in the SSL peer certificate. 2 to check the existence of a common name and also verify that it matches the hostname provided. In production environments the value of this option should be kept at 2 (default value).

Well-intentioned developers not only routinely misunderstand these parameters, but often set `CURLOPT_SSL_VERIFY HOST` to TRUE, thereby changing it to 1 and thus accidentally disabling hostname verification with disastrous consequences (see Section 7.1).

PHP. PHP provides several methods for establishing SSL connections. For example, **`fsockopen`**, which opens a raw socket to the remote server, can be used to connect to SSL servers by putting "ssl://" in the URL. Even though **`fsockopen`** does not perform any certificate checks whatsoever, PHP application developers routinely use it for SSL connection establishment (see Section 9).

PHP also provides a cURL binding, which uses cURL's default settings to establish SSL connections with proper certificate validation. As we show in Sections 7.1, 7.2, and 7.3, application developers often set cURL options incorrectly, overriding the defaults and breaking certificate validation.

Python. Several Python modules can be used for SSL connection establishment. *urllib*, *urllib2*, and *httplib* connect to SSL servers but do not check certificates. This is clearly documented in a bright pink box on the *urllib* front page:[6]

> Warning: When opening HTTPS URLs, it does not attempt to validate the server certificate. Use at your own risk!

Nevertheless, even high-security applications routinely use these modules for SSL connection establishment (see Section 9).

Python also has an *ssl* module. This module verifies the certificate's chain of trust, but not the hostname. The application must do its own hostname verification. In Python version 3, the *ssl* module introduced the *match_hostname* method for hostname verification, but it must be explicitly called by the application.

[2] `http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`

[3] `http://hc.apache.org/httpcomponents-client-ga/`

[4] `http://code.google.com/p/weberknecht/`

[5] `http://curl.haxx.se/`

[6] `http://docs.python.org/library/urllib.html`

5. SSL IN NON-BROWSER SOFTWARE

We analyze a representative sample of non-browser software applications and libraries that use SSL for secure Internet connections. Some programs, such as instant messenger clients and simple mobile banking apps, are fairly straightforward in their use of SSL. Others, especially middleware libraries, use SSL as part of a multi-layer software stack. Many of the programs we analyze transmit extremely sensitive data—private files of individual users in the case of cloud clients, financial information of customers in the case of merchant SDKs, developer account credentials in the case of mobile advertising software—over potentially insecure public networks, thus it is absolutely critical that they use SSL correctly.

Cloud client APIs. As cloud-computing platforms such as Amazon EC2 grow in popularity, their operators supply client SDKs through which third-party software can transmit user data to cloud-based storage, manage cloud-based computation (e.g., start and terminate virtual instances), and access other cloud services. For example, Amazon provides EC2 API tools in Java, PHP, Python, and Perl. Apache Libcloud is an example of an independent library for accessing multiple cloud providers.

Merchant payment SDKs. Operators of e-commerce websites often rely on third parties such as PayPal and Amazon Flexible Payments Service (FPS) to process their customers' payments. Payment processors provide merchant SDKs (software development kits) in a variety of languages. These libraries are designed to be integrated into the back end of e-commerce websites. Merchant software uses them to transmit customers' payment details and/or receive notifications when payments are made by customers.

An online store typically has two options for payment processing. The first option is to have the customer enter payment details directly into the payment processor's website. When the customer checks out, the merchant's website redirects her browser to PayPal or Amazon, where the customer enters her name, credit or debit card number, etc. The merchant never sees these details. Once the payment is complete, the payment processor redirects the customer back to the merchant's website and notifies the merchant.

The merchant's site runs a daemon listening for IPN (Instant Payment Notification) calls from the payment processor. Upon receiving a notification, the merchant is advised to verify that the call indeed originated from the processor (some merchants skip this step, opening the door to "shop-for-free" attacks [23]). The merchant then completes the transaction with the customer.

The second option is preferred by larger, more established stores. It does not require the customer to leave the merchant's website and allows the merchant to collect payment details directly from the customer. The back-end software on the merchant's website then transmits these details to the payment processor's gateway over an SSL connection and receives the confirmation that the payment succeeded (often over the same SSL connection).

Fig. 4 shows schematically the interaction between the merchant's server and the payment gateway. The SSL client is the merchant's back-end software (running on the merchant's *server*), while the payment gateway acts as the SSL server.

We analyzed SSL connection establishment in popular merchant SDKs, including Java and PHP SDKs for Amazon Flexible Payments Service and multiple interfaces to PayPal: Payments Pro, Transactional Information, and Mass Pay (all in Java), as well as Payments Standard and Invoicing (PHP). We also analyzed both Amazon's and PayPal's utilities that merchants can use to verify the origin of IPN (Instant Payment Notification) calls.

We also analyzed several open-source shopping carts written in PHP: osCommerce, ZenCart, Ubercart, and PrestaShop. Shopping

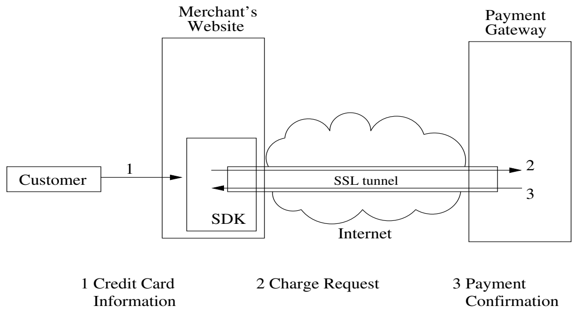

Figure 4: Merchant SDK interacting with payment processor.

carts are an important component of e-commerce websites. They keep track of customers' shipping and billing information and allow them to purchase multiple items in one transaction. When the customer checks out, the shopping cart generates a summary of the purchases and the total price and sends it to the payment gateway. Shopping carts include modules for many payment processors.

Web-services middleware. Many modern programs rely on Web services. A Web service is "a software system designed to support interoperable machine-to-machine interaction over a network."[7] A service has an interface described in a machine-readable XML format. Different providers may provide different concrete implementations of this interface. Other systems interact with the service by sending and receiving messages.

Messages to and from Web services are sent using XML-based Simple Object Access Protocol (SOAP) or REpresentational State Transfer (REST). From the viewpoint of the client software, a Web service can be thought of as providing a remote procedure call (RPC) interface, while SOAP or REST middleware marshals and unmarshals arguments of RPC calls.

To interact with such a Web service—for example, if a cloud client implemented in Java wants to interact with Amazon EC2 — existing Java software often uses SOAP middleware such as Apache Axis, Axis 2, or Codehaus XFire (see Section 8). Similarly, if an Android app needs real-time "push" notifications, it may use a client-side library to connect to the REST-based Pusher service.[8]

These middleware frameworks are responsible for transmitting Web-service messages over the network. If the connection must be secure, the middleware typically uses SSL but delegates actual SSL connection management to a data-transport library such as Apache HttpClient or Weberknecht (see Section 4.2).

Mobile advertising. Mobile advertising services such as AdMob supply software that providers of mobile apps install on their sites. When a new app instance is initialized on a customer's phone, it connects to the provider's site, which in turn notifies the AdMob server so that all ads shown to this app instance will be associated with the provider's account (to enable ad revenue sharing, etc.). The connection from the app provider's site to the AdMob server contains the provider's credentials and must be protected by SSL.

6. EXPERIMENTAL TESTBED

Our primary methodology for the initial discovery of SSL certificate validation bugs is black-box fuzzing. We test applications and libraries implementing SSL client functionality on two Dell laptops running Microsoft Windows 7 Professional Service Pack

[7]`http://www.w3.org/TR/ws-arch/`
[8]`http://pusher.com`

1 and Ubuntu Linux 10.04, respectively. Mobile applications are tested on a Nexus One smartphone running Android 2.3.6 and an iPad 2 running iOS 4.2.1.

We use local DNS cache poisoning to divert clients' connections to a simulated attack server executing on an old Dell laptop with Ubuntu Linux 10.04. To simulate a man-in-the-middle attacker, we built two prototypes: one in Java, using JKS keystore to manage the attacker's certificates and keys, the other in C, using OpenSSL for certificate and key management. We also used Fiddler, a Web debugging proxy [9]. If Fiddler encounters a connection request to a server it has not seen before, it creates a new certificate with the common name matching the requested name and stores it in its repository; otherwise, it retrieves an existing certificate from its repository. Fiddler then presents the certificate to the client, allowing us to simulate a man-in-the-middle attacker who presents self-signed certificates with correct common names. In addition, we enabled Fiddler to capture and decrypt HTTPS connections.

Our simulated "man-in-the-middle" server presents the client with several certificates: (1) a self-signed certificate with the same common name as the host the client is attempting to connect to, (2) a self-signed certificate with an incorrect common name, and (3) a valid certificate issued by a trusted certificate authority to a domain called `AllYourSSLAreBelongTo.us`. If the client establishes an SSL connection, the attack server decrypts traffic sent by the client. It can then establish its own SSL connection to any legitimate server specified by the attacker and forward the client's traffic. The attack server also listens for the legitimate server's response, decrypts and logs it, re-encrypts it with the symmetric key the attacker shares with the client and forwards it to the client.

If we observed a particular client successfully establishing an SSL connection when presented with any of the attack certificates, we analyzed the source code of the client or, in the case of closed-source applications, the results of reverse-engineering, decompilation, and runtime traces (focusing in particular on calls to SSL libraries) in order to find the root cause of the vulnerability.

In Sections 7 through 10, we describe the vulnerabilities in specific programs, arranged by error type.

7. MISUNDERSTANDING THE SSL API

7.1 Amazon Flexible Payments Service (PHP)

Amazon Flexible Payments Service (FPS) provides SDKs that merchants use to transmit customers' payment details to the FPS gateway. The PHP version of the FPS SDK uses a wrapper around the *libcurl* library (see Section 4.2) to establish an SSL connection to the gateway. cURL's options for certificate validation are set in `src\Amazon\FOPS\Client.php` as follows:

```
curl_setopt($curlHandle, CURLOPT_SSL_VERIFYPEER, true);
curl_setopt($curlHandle, CURLOPT_SSL_VERIFYHOST, true);
...
// Execute the request
$response = curl_exec($curlHandle);
```

This well-intentioned code contains a fatal mistake. cURL's default value of `CURLOPT_SSL_VERIFYHOST` is correctly set to 2. In the `curl_setopt($curlHandle,CURLOPT_SSL_VERIFYHOST, true)` call, **`true`** silently turns into 1, overriding the default and instructing cURL to check the existence of *any* common name in the certificate (Section 4.2), which may or may not match the name requested.

Any PHP code using this Amazon-provided SDK to establish an SSL connection to the Amazon Flexible Payments Service gateway is insecure against a man-in-the-middle attack.

The URL verification utility—found in `src\Amazon\IpnReturnUrlValidation\SignatureUtilsForOutbound.php`—is broken in a very similar way. This utility is critically important because it is used by merchants to verify the origin of the calls informing them that a customer's payment has been successfully processed (see Section 5). Because Amazon's PHP SDK does not correctly verify the origin of the IPN call, e-commerce sites using it may be vulnerable to "shop-for-free" attacks [23].

7.2 PayPal Payments Standard and PayPal Invoicing (PHP)

PayPal Payments Standard SDK implemented in PHP uses cURL. The previous version disabled all certificate validation checks:

```
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, FALSE);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, FALSE);
```

The version released on April 27, 2012, "fixes" the problem:

```
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, TRUE);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, TRUE);
```

As in Section 7.1, this code overrides the correct default value of `CURLOPT_SSL_VERIFYHOST` and breaks hostname verification.

PayPal Invoicing contains similarly broken code:

```
public function setHttpTrustAllConnection(
    $trustAllConnection)
{
  $this->curlOpt[CURLOPT_SSL_VERIFYPEER] =
      !$trustAllConnection;
  $this->curlOpt[CURLOPT_SSL_VERIFYHOST] =
      !$trustAllConnection;
}
```

Any PHP code using these PayPal-provided SDKs to establish an SSL connection to PayPal's payment processing gateway is insecure against a man-in-the-middle attack.

7.3 PayPal IPN in ZenCart

ZenCart's functionality for PayPal IPN shows a profound misunderstanding of cURL's parameters. It disables certificate validation entirely, yet attempts to enable hostname verification—even though the latter has no effect if certificate validation is disabled.

```
$curlOpts=array( ...
          CURLOPT_SSL_VERIFYPEER => FALSE,
          CURLOPT_SSL_VERIFYHOST => 2
          ... );
```

7.4 Lynx

Lynx is a text-based browser, included in our study because it is often used programmatically by other software. It relies on GnuTLS to validate SSL certificates:

```
ret = gnutls_certificate_verify_peers2(handle->gnutls_
     state, &tls_status);
if (ret < 0) {
  int flag_continue = 1;
  char *msg2;

  if (tls_status & GNUTLS_CERT_SIGNER_NOT_FOUND) {
    msg2 = gettext("no issuer was found");
  } else if (tls_status & GNUTLS_CERT_SIGNER_NOT_CA) {
    msg2 = gettext("issuer is not a CA");
  } else if (tls_status & GNUTLS_CERT_SIGNER_NOT_FOUND)
      {
    msg2 = gettext("the certificate has no known issuer"
        );
  } else if (tls_status & GNUTLS_CERT_REVOKED) {
    msg2 = gettext("the certificate has been revoked");
  } else {
    msg2 = gettext("the certificate is not trusted"); }
        ... }
```

This code misinterprets the semantics of `gnutls_certificate_verify_peers2`. As explained in Section 4.1, this function indeed sets the `tls_status` code if certificate validation fails, but for certain errors—including self-signed certificates!—it returns 0. Even though the above code includes *two* identical checks for `GNUTLS_CERT_SIGNER_NOT_FOUND`, neither check is ever executed when `GNUTLS_CERT_SIGNER_NOT_FOUND` is actually true! In this case hostname verification is correct, but chain-of-trust verification is broken.

7.5 Apache HttpClient

The most widely used version of Apache HttpClient is 3.1, released in 2007. This library, as well as its earlier versions, sets up SSL connections using JSSE's `SSLSocketFactory` without performing its own hostname verification (see Sections 4.1 and 4.2). As a consequence, Apache HttpClient 3.* accepts any certificate with a valid chain of trust, regardless of the name. As mentioned in Section 4.2, the same bug occurs in Weberknecht.

The hostname verification bug in HttpClient was fixed in version 4.0-alpha1 [1]. The current version, 4.2.1, has its own hostname verifier and delegates chain-of-trust verification to JSSE. Unfortunately, as we show in Section 8, the existence of a correct implementation of HttpClient has had little effect on the security of applications that rely on HttpClient for SSL connection establishment. Apache HttpClient 4.* involved a major architectural re-design, thus much of legacy and even new software still relies on version 3.*. The use of HttpClient is often hidden inside Web-services middleware such as Axis 2 and XFire, which—several years after version 4.* became available—still ship with HttpClient 3.* and thus skip hostname verification for SSL certificates.

It is worth noting that the custom hostname verification code added to HttpClient 4.* is incorrect and will reject valid certificates. The following code is from HttpClient 4.2.1:

```
// The CN better have at least two dots if it wants
       wildcard
// action.  It also can't be [*.co.uk] or [*.co.jp] or
// [*.org.uk], etc...
String parts[] = cn.split("\\.");
boolean doWildcard = parts.length >= 3 &&
          parts[0].endsWith("*") &&
          acceptableCountryWildcard(cn) &&
          !isIPAddress(host);
if(doWildcard) {
  if (parts[0].length() > 1) { // e.g. server*
    String prefix = parts[0].substring(0, parts.length
        -2); // e.g. server
    String suffix = cn.substring(parts[0].length());
        // skip wildcard part from cn
    String hostSuffix = hostName.substring(prefix.length
        ()); // skip wildcard part from host
    match = hostName.startsWith(prefix) && hostSuffix.
        endsWith(suffix);
  } else {
    match = hostName.endsWith(cn.substring(1));
    }
  if(match && strictWithSubDomains) {
    // If we're in strict mode, then [*.foo.com] is not
    // allowed to match [a.b.foo.com]
    match = countDots(hostName) == countDots(cn);
    }
} else {
  match = hostName.equals(cn);
}
```

This code computes the length of the prefix by subtracting 2 from the number of *parts* (determined by the number of dots in the name). This logic is incorrect: validity of the first part of a domain name should have nothing to do with the total number of parts. For example, it will reject `mail.<a>.<b>.com` if the name in the certificate is `m*.<a>.<b>.com`.

Furthermore, the original patch, as well as its derivatives, has a minor bug in the regular expression for parsing IPv4 addresses, causing it to accept IP addresses starting with zero (this does not immediately result in a security vulnerability):

```
private static final Pattern IPV4_PATTERN =
       Pattern.compile("^(25[0-5]|2[0-4]\\d|[0-1]?\\d
            ?\\d)(\\.(25[0-5]|2[0-4]\\d|[0-1]?\\d?\\d))
            {3}\$");
```

7.6 Trillian

Trillian, a popular instant messenger client, relies on OpenSSL for SSL connection establishment. By default, OpenSSL does not throw a run-time exception if the certificate is self-signed or has an invalid chain of trust. Instead, it sets flags. Furthermore, OpenSSL does not provide any hostname verification.

If the application has called `SSL_CTX_set` to set the `SSL_VERIFY_PEER` flag (see Section 4.1), then `SSL_connect` exits and prints an error message when certificate validation fails. Trillian does not set the `SSL_VERIFY_PEER` flag. When this flag is not set, `SSL_connect` returns 1. The application is then expected to check the status of certificate validation by calling `SSL_get_verify_result`. Trillian does not call this function.

Trillian thus accepts any SSL certificate and is insecure against a man-in-the-middle attack. Depending on the specific module chosen by the Trillian user, this reveals usernames, passwords, security tokens, etc. for Google Talk (typically compromising all of the user's Google services), AIM, ICQ, Windows Live (including SkyDrive), and Yahoo! Messenger (and thus all Yahoo! services).

Interestingly, it was reported back in 2009 that older versions of Trillian do not correctly validate MSN certificates [20]. This bug was ostensibly fixed in Trillian 4.2. Our analysis shows, however, that SSL certificate validation is still completely broken for all services, not just for MSN (Windows Live), in Trillian 5.1.0.19.

7.7 Rackspace

The Rackspace app for iOS (version 2.1.5) is an open-source application for administering Rackspace cloud services. It uses the OpenStack iOS cloud client framework, which in turn relies on the ASIHTTPRequest library to set up HTTPS connections.

ASIHTTPRequest provides a configuration variable `ValidatesSecureCertificate`, set to 1 by default. If reset to 0, it turns off both chain-of-trust and hostname verification. OpenStack supports multiple accounts on remote servers and lets users customize SSL certificate validation on per-account basis using the `ignoreSSLValidation` variable. The value of this variable depends on the GUI switch `validateSSLSwitch`, which should be shown to the user.

The Rackspace app (version 2.1.5) does not present the user with this option.[9] The GUI switch `validateSSLSwitch` is thus never displayed or explicitly assigned. Instead, it is simply initialized to 0 by the Objective-C allocator. This turns on `ignoreSSLValidation` in ASIHTTPRequest, which in turn sets `ValidatesSecureCertificate` to 0 and disables certificate validation.

As a consequence, SSL connections established by the Rackspace app on iOS are insecure against a man-in-the-middle attack.

7.8 TextSecure

TextSecure is an Android application for encrypting SMS and MMS messages. It was written by Moxie Marlinspike who had

[9] We are informed by Mike Mayo that this was an accidental oversight and will be fixed in subsequent releases of the app.

previously discovered several SSL certificate validation vulnerabilities [11, 12]. This following code can be found in the application (however, it does not appear to be reachable from the user interface and may not currently lead to an exploitable vulnerability):

```
schemeRegistry.register(new Scheme("http",
    PlainSocketFactory.getSocketFactory(), 80));
schemeRegistry.register(new Scheme("https",
    SSLSocketFactory.getSocketFactory(), 443));
...
HttpHost target = new HttpHost(hostUrl.getHost(),
    hostUrl.getPort(), HttpHost.DEFAULT_SCHEME_NAME);
...
HttpResponse response = client.execute(target, request);
```

Even if the port number is 443, `DEFAULT_SCHEME_NAME` is "http" and the connection is over HTTP, not HTTPS.

8. USING INSECURE MIDDLEWARE

As explained in Section 5, software based on Web services usually relies on middleware libraries to manage network connections. SSL functionality inside these libraries is opaque to the applications. If the middleware employs a broken HTTPS implementation that does not correctly validate SSL certificates, all applications based on it typically "inherit" the vulnerability.

8.1 Apache Axis, Axis 2, Codehaus XFire

Apache Axis is an open-source Java implementation of SOAP. The latest release is 1.4, discontinued in 2007 in favor of Axis 2, but the library is still used, for example, in PayPal's Java SDKs. Apache Axis 2 is a complete redesign of Apache Axis. The latest release is 1.6.2. Codehaus XFire is another open-source Java implementation of SOAP. It was discontinued in 2007 in favor of Apache CXF, but is still used, for example, in Amazon's EC2 Java SDK. The latest release of XFire is 1.2.6.

Apache Axis uses its own version of HttpClient, while Axis 2 and XFire use Apache HttpClient version 3.1. Both versions of HttpClient rely on `SSLSocketFactory` for SSL connection establishment but mistakenly omit hostname verification (Section 4.2).

SSL vulnerabilities caused by bugs in Web-services middleware are pervasive in Amazon libraries. Affected software includes **Amazon EC2 API Tools** Java library, which uses XFire to set up SSL connections to EC2 servers, and **Amazon Flexible Payments Service (Java)** merchant SDK, which relies on an old Apache HttpClient. The latter library is used by merchants to transmit customers' payment details to the FPS gateway. The PHP version of the library is broken, too, but in a very different way (Section 7.1). In contrast to the PHP version, however, the Java utility for verifying instant payment notifications uses JSSE's `HttpsClient` instead of Apache HttpClient and thus checks SSL certificates correctly.

Other software that relies on Axis includes Java SOAP SDKs for **PayPal Payments Pro (Direct Payment)**, **PayPal Transactional Information**, and **PayPal Mass Pay**, as well as **Apache ActiveMQ** implementation of JMS (Java Message Service).

8.2 Pusher

Pusher is a WebSocket-based API that provides real-time messaging functionality to mobile and Web applications. Pusher's Android libraries[10] are based on Weberknecht (see Section 4.2). Any application using these libraries (e.g., GitHub's Gaug.es) is thus insecure. It is also worth noting that Gaug.es is using an updated version of Weberknecht, which, in addition to incorrectly using raw `SSLSocketFactory` (see Section 4.1), disables the trust manager.

[10]`https://github.com/EmoryM/Android_Pusher`

In summary, **any software using any of the above Web-services frameworks is insecure against a man-in-the-middle attack**.

Apache CXF. Apache CXF is a continuation of XFire. It supports SOAP, along with REST and CORBA; the latest release is 2.6.1. It does not rely on Apache HttpClient. Instead, SSL connections are established using OpenJDK's `HttpsClient`. Therefore, properly configured instances of CXF do verify hostnames.

Apache CXF provides an application-controlled option to turn off certificate validation. Certificate validation is enabled by default, but was disabled in the sample `wsdl_first_https` code supplied with CXF until we notified the developers.

9. USING INSECURE SSL LIBRARIES

As described in Section 4.2, PHP's **`fsockopen`** does not validate SSL certificates. Nevertheless, it is often used even by applications that must be secure against a man-in-the-middle attack. For example, **PayPal's IPN** utility contains this code:

```
// post back to PayPal utility to validate
...
$fp = fsockopen ('ssl://www.paypal.com', 443, $errno,
     $errstr, 30);
```

This code is replicated in PayPal payment modules for **ZenCart** and **PrestaShop** shopping carts. PrestaShop uses **`fsockopen`** in its CanadaPost payment module, too. Other similarly vulnerable software includes **Open Source Classifieds**,

Python's URL libraries do not validate certificates (Section 4.2), yet developers still use them for SSL connections. Examples include **Tweepy**, a library for accessing Twitter API that uses *httplib*, and Mozilla's **Zamboni** project, which accepts contributions for extension developers and uses *urllib2* to connect to PayPal.

10. BREAKING OR DISABLING CERTIFICATE VALIDATION

In general, disabling proper certificate validation appears to be the developers' preferred solution to any problem with SSL libraries. Here are a few typical quotes from developers' forums:

- "I want my client to accept any certificate (because I'm only ever pointing to one server) but I keep getting a `javax.net.ssl.SSLException: Not trusted server certificate exception`"[11]—*[note the fallacious reasoning!]*
- "Tutorial: Disabling Certificate Validation in an HTTPS Connection... Reply: Thank you very much. You solved my biggest problem in the project." [12]
- "I have always turned off CURLOPT_SSL_VERIFYPEER in curl."[13]
- "I am using axis on java to consume a webservice. The web service is in https, and I want to avoid the the check for certificate."[14]

[11]`http://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https`
[12]`http://www.exampledepot.com/egs/javax.net.ssl/trustall.html`
[13]`http://stackoverflow.com/questions/10102225/curl-ssl-certificates`
[14]`http://stackoverflow.com/questions/9079298/axis-fake-certificate`

- "However, by default, SSL support in NSStream is a little paranoid. It won't, for example, use a self-signed certificate or an expired certificate to establish a secure connection. NSStream does a number of validity checks when establishing the secure connection, and if they don't all pass, the streams appear to be valid, but no data gets sent or received. This is somewhat frustrating, and it could be there's a way to find out when the secure connection failed, but I haven't been able to find it in the documentation, or using Google. There is an error domain declared for these errors (NSStreamSocketSSLErrorDomain), but in my experimentation, no errors gets generated, the streams even accept bytes for transfer, but nothing happens." [15]

Unfortunately, these bad development practices find their way even into critical software responsible for transmitting financial information and sensitive data, where security against man-in-the-middle attacks is absolutely essential and SSL certificate validation should be mandatory. For example, a comment in the Authorize.Net eCheck module of ZenCart says that certificate validation is disabled for "compatibility for SSL communications on some Windows servers (IIS 5.0+)"—note the fallacious reasoning!

10.1 Chase mobile banking

Chase is a major US bank. SSL connections established by its mobile banking application on Android are insecure against a man-in-the-middle attack. This allows a network attacker to capture credentials, such as username and password, of any Chase customer using this app, along with the rest of their session.

Decompilation and analysis of this app's code show that it overrides the default `X509TrustManager`. The replacement code simply returns without checking the server's certificate. The code below is the result of reverse-engineering, thus variable names and other details may differ from the actual code:

```
public final void checkServerTrusted(X509Certificate[]
      paramArrayOfX509Certificate, String paramString)
{
  if ((paramArrayOfX509Certificate != null) && (
       paramArrayOfX509Certificate.length == 1))
    paramArrayOfX509Certificate[0].checkValidity();
  while (true)
  {
    return;
    this.a.checkServerTrusted(
         paramArrayOfX509Certificate, paramString);
  }
}
```

Note the unreachable invocation of `checkServerTrusted`. We conjecture that this was a temporary plug during development that somehow found its way into the production version of the app.

10.2 Apache Libcloud

Apache Libcloud[16] is a Python library extension providing support for 26 different cloud service providers. Libcloud relies on the underlying Python library to verify the chain of trust in SSL certificates; internally, Python uses OpenSSL. Once the chain of trust is verified, Libcloud verifies the hostname using the `_verify_hostname` method in `httplib_ssl.py`. This code uses an incorrect regular expression for hostname verification. For example, it accepts `oogle.com` as a match for `google.com`, exposing all Libcloud clients to a man-in-the-middle attack:

```
def _verify_hostname(self, hostname, cert):
  # Verify hostname against peer cert
  # Check both commonName and entries in subjectAltName,
  # using a rudimentary glob to dns regex check
  # to find matches

  common_name = self._get_common_name(cert)
  alt_names = self._get_subject_alt_names(cert)

  # replace * with alphanumeric and dash
  # replace . with literal .
  valid_patterns = [re.compile(pattern.replace(r".", r"
      \.").replace(r"*", r"[0-9A-Za-z]+"))
    for pattern
      in (set(common_name) | set(alt_names))
  ]

  return any(
    pattern.search(hostname)
    for pattern in valid_patterns
  )
```

This bug has been fixed in Libcloud version 0.11.1 after we notified the developers.

10.3 Amazon Elastic Load Balancing API Tools

This library overrides JDK's default `X509TrustManager` to disable hostname verification. Even if `X509TrustManager` had not been overriden, this library employs Codehaus XFire which does not perform hostname verification (see Section 8.1).

10.4 Shopping carts

osCommerce, **ZenCart**, **Ubercart**, and **PrestaShop** are open-source shopping carts implemented in PHP. They use cURL for SSL connections to payment gateways. If cURL is not available, they typically fall back on (insecure) **`fsockopen`**.

All carts are bundled with plugin modules for specific payment processors. Almost without exception, these modules turn off certificate validation. In ZenCart, vulnerable modules include LinkPoint, Authorize.Net, and PayPal Payments Pro, as well as PayPal IPN functionality (see Section 7.3). The insecure LinkPoint module contains an amusing comment at the beginning of the file: "### YOU REALLY DO NOT NEED TO EDIT THIS FILE! ###"

Vulnerable modules include eBay, PayPal, and Canada Post in PrestaShop, PayPal, Authorize.Net, and CyberSource in Ubercart, Sage Pay Direct, Authorize.Net, MoneyBookers, and PayPal Express, Pro, Pro PayFlow, and Pro PayFlow EC in osCommerce.

SSL connections to payment gateways from merchants using any of these carts are insecure against a man-in-the-middle attack.

The only exceptions are Google modules for PrestaShop and osCommerce. The Google Checkout module for osCommerce comes from `code.google.com` and is not bundled with osCommerce. It sets `CURLOPT_SSL_VERIFYPEER` to **`true`** and leaves `CURLOPT_SSL_VERIFYHOST` to its correct default value, 2. By contrast, the official, PayPal-provided PayFlow module disables certificate validation.

10.5 AdMob

Google's AdMob provides sample code to mobile site owners that they can use on their servers to associate instances of their mobile apps with their developer accounts (see Section 5). This code uses cURL to establish an SSL connection to AdMob's server, but turns off certificate validation. A man-in-the-middle attacker can thus gain access to all of the developers' Google services.

10.6 Android apps

Groupon Redemptions, an Android app for merchants, disables certificate validation *twice*: by allowing any hostname via the "allow all" hostname verifier and by binding to an empty trust man-

[15] `http://iphonedevelopment.blogspot.com/2010/05/nsstream-tcp-and-ssl.html`
[16] `http://libcloud.apache.org/`

ager. Similarly, **Breezy**, an app for secure document printing, disables hostname verification and overrides the default trust manager.

ACRA, an Android library for posting application crash reports to a Google Doc, overrides the default trust manager. Any app using this library is insecure against a man-in-the-middle attack.

10.7 AIM

AIM client version 1.0.1.2 on Windows uses Microsoft's CryptoAPI. Runtime analysis shows that it calls CryptoAPI's certificate validation function `CertVerifyCertificateChainPolicy`. To disable certificate validation, it passes a `CERT_CHAIN_POLICY_PARA` variable with `CERT_CHAIN_POLICY_ALLOW_UNKNOWN_CA_FLAG` set, instructing CryptoAPI to accept certificates signed by untrusted authorities. AIM does not perform any hostname verification, either.

10.8 FilesAnywhere

FilesAnywhere is an application for managing cloud storage. It uses CryptoAPI for SSL connections and accepts both self-signed and third-party certificates.

FilesAnywhere has an interesting peculiarity. If presented with a Google certificate when it attempts to connect to a non-Google server, it shows a warning message "The WebDav server has a new address. Please specify http://google.com in the profile." If presented with any other third-party certificate, it silently accepts it and sends user's data to a wrong, potentially malicious server.

11. OUR RECOMMENDATIONS

Whenever application developers must deal with SSL, the conventional advice is to use standard SSL libraries. This advice is correct, but insufficient. As this paper shows, even developers of high-security software often use standard SSL libraries incorrectly. The following recommendations are informed by our analyses of broken SSL certificate validation in diverse applications.

11.1 For application developers

DO use fuzzing (black-box, if necessary) and adversarial testing to see how the application behaves when presented with abnormal SSL certificates. Even when the vulnerabilities are subtle, the symptoms usually are not. In many of our case studies, it is obvious that *the software in question has never been tested* with any certificates other than those of the intended server. When presented with a certificate issued to `AllYourSSLAreBelongTo.us` instead of the expected Amazon or PayPal or Chase certificate, these programs eagerly establish SSL connections and spill out their secrets. These vulnerabilities should have manifested during testing.

DON'T modify application code and disable certificate validation for testing with self-signed and/or untrusted certificates. We found in our case studies that developers forget to reverse these modifications even for the production version of the software. Instead, create a temporary keystore with the untrusted CA's public key in it. While testing your code with self-signed or untrusted certificates, use that keystore as your trusted keystore.

DON'T depend on the library's defaults to set up the SSL connection securely. Default settings can and do change between different libraries or even different versions of the same library—for example, cURL prior to version 7.10 did not validate certificates by default, but version 7.10 and later do. Always explicitly set the options necessary for secure connection establishment.

11.2 For SSL library developers

DO make SSL libraries more explicit about the semantics of their APIs. In many cases, it is obvious that application developers do not understand the meaning of various options and parameters. For example, the PHP libraries for Amazon Flexible Payments Services and PayPal Payments Standard attempt to enable hostname verification in cURL, but instead accidentally override the correct default value and end up disabling it (Sections 7.1 and 7.2). This shows that even safe defaults may be insufficient. Lynx attempts to check for self-signed certificates, but misinterprets the meaning of return values of GnuTLS's certificate validation function and the check is never executed (Section 7.4). Formalizing the precise semantics of SSL library API and rigorously verifying the "contracts" between the application and the library is an interesting topic for future research and may call for programming language support.

DON'T delegate the responsibility for managing SSL connections to the applications. Existing SSL libraries expose many options to higher-level software. This is fraught with peril. Application developers may not realize that they must explicitly choose certain options in order to enable certificate validation. Therefore, libraries should use safe defaults as much as possible. Furthermore, they should not silently skip important functionality such as hostname verification as JSSE does when the algorithm field is NULL or an empty string (see Section 4.1). Instead, they should raise a runtime exception or inform the application in some other way.

DO design a clean and consistent error reporting interface. Libraries such as OpenSSL and GnuTLS report some errors via return values of functions, while other errors from the same function are reported through a flag passed as an argument. Inconsistent interfaces confuse developers who then mistakenly omit some error checks in their applications.

These recommendations provide short-term fixes. A principled solution to the problem must involve a complete redesign of the SSL libraries' API. Instead of asking application developers to manage incomprehensible options such as `CURLOPT_SSL_VERIFYPEER` or `SSL_get_verify_result`, they should present high-level abstractions that explicitly express security properties of network connections in terms that are close to application semantics: for example, a "confidential and authenticated tunnel." The library should also be explicit about the security consequences of any application-controlled option: for example, instead of "verify hostname?", it could ask "Anyone can impersonate the server. Ok or not?"

12. RELATED WORK

Independently of this work, Kevin McArthur announced multiple vulnerabilities caused by improper SSL certificate validation in PHP software. [17] Affected programs include, among others, osCommerce, Ubercart, PrestaShop, and three PayPal SDKs.

Moxie Marlinspike demonstrated several vulnerabilities in certificate validation code in browsers and SSL libraries, including the lack of basic constraint checking (e.g., checking the CA bit) [11] and incorrect parsing of NULL characters in the "CommonName" field [12]. By contrast, we focus on non-browser software that uses (mostly) correct SSL libraries incorrectly.

Kaminsky et al. [10] showed that parsing differences between CA software and browser certificate validation code can result in a CA issuing a certificate that can be used for a man-in-the-middle attack. By contrast, we investigate certificate validation bugs in non-browser clients, not in CA software.

Stevens et al. showed how an attacker can leverage MD5 hash collisions to get a CA to issue a specially crafted certificate that is valid for an ordinary host but whose hash collides with that of a certificate for a new, rogue intermediate CA [18]. By contrast, our attacks do not involve certificate forgery.

[17] `http://www.unrest.ca/peerjacking`

Several certificate authorities such as Comodo [5] and DigiNotar [6] were recently compromised and used by attackers to issue fake certificates for popular websites. By contrast, our attacks do not involve CA compromise.

To mitigate the risks of rogue certificates, Evans et al. proposed certificate pinning, i.e., pre-established bindings in the browser between well-known websites and their certificates [8]. Certificate pinning is not supported by any of the software we analyzed.

Several large-scale studies analyzed HTTPS deployment [7, 22] and found many errors in SSL certificates. One of the most common errors is a mismatch between the server's fully qualified domain name and certificate's identifiers. This misconfiguration alone does not enable a man-in-the-middle attack.

Chen et al. showed how a malicious proxy can exploit browser bugs for man-in-the-middle attacks on HTTPS [3]. By contrast, our attacks do not depend on browser bugs.

Side-channel attacks can extract information from encrypted traffic even when SSL is correctly deployed [4, 19]. By contrast, we found vulnerabilities that enable a man-in-the-middle attacker to decrypt SSL traffic, obviating the need for side-channel analysis. Other side-channel attacks include a timing attack that extracts the private key from OpenSSL implementations [2].

13. CONCLUSION

The main lesson of this paper is that using SSL in non-browser software is a surprisingly challenging task. We demonstrated that even applications that rely on standard SSL libraries such as JSSE, OpenSSL, GnuTLS, etc. often perform SSL certificate validation incorrectly or not at all. These vulnerabilities are pervasive in critical software, such as Amazon FPS and PayPal libraries for transmitting customers' payment details from merchants to payment gateways; integrated shopping carts; Amazon EC2, Rackspace, and other clients for remote administration of cloud storage and virtual cloud infrastructure; Chase mobile banking on Android; and many other popular programs. Their SSL connections are completely insecure against a man-in-the-middle attack.

We also presented our recommendations for safer use of SSL in non-browser software. Future research directions include (1) development of better black-box testing and code analysis tools for discovering errors in SSL connection establishment logic, (2) design of formal verification techniques and programming language support for automatically checking whether applications use SSL libraries correctly and not misinterpret the meaning of critical options and parameters, and (3) design of better APIs for SSL and other secure networking protocols.

Acknowledgments. This research was partially supported by the NSF grants CNS-0331640, CNS-0746888, and CNS-0905602, two Google research awards, Samsung, and the MURI program under AFOSR Grant No. FA9550-08-1-0352. Boneh also thanks iSEC partners for helpful conversations about this work.

We acknowledge Amazon, Apache, Chase, GitHub, Lynx, PayPal, and Rackspace developers for recognizing and promising to repair the vulnerabilities after they were brought to their attention. We thank Colm O hEigeartaigh for explaining the intended behavior of certificate validation in Apache CXF.

References

[1] https should check CN of x509 cert. `https://issues.apache.org/jira/browse/HTTPCLIENT-613`.

[2] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security*, 2003.

[3] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang. Pretty-Bad-Proxy: An overlooked adversary in browsers' HTTPS deployments. In *S&P*, 2009.

[4] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in Web applications: A reality today, a challenge tomorrow. In *S&P*, 2010.

[5] Comodo report of incident. `http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html`, 2011.

[6] Diginotar issues dodgy SSL certificates for Google services after break-in. `http://www.theinquirer.net/inquirer/news/2105321/diginotar-issues-dodgy-ssl-certificates-google-services-break`, 2011.

[7] P. Eckersley and J. Burns. An observatory for the SSLiverse. In *DEFCON*, 2010.

[8] C. Evans and C. Palmer. Certificate pinning extension for HSTS. `http://www.ietf.org/mail-archive/web/websec/current/pdfnSTRd9kYcY.pdf`, 2011.

[9] Fiddler - Web debugging proxy. `http://fiddler2.com/fiddler2/`.

[10] D. Kaminsky, M. Patterson, and L. Sassaman. PKI layer cake: new collision attacks against the global X.509 infrastructure. In *FC*, 2010.

[11] Moxie Marlinspike. IE SSL vulnerability. `http://www.thoughtcrime.org/ie-ssl-chain.txt`, 2002.

[12] Moxie Marlinspike. Null prefix attacks against SSL/TLS certificates. `http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf`, 2009.

[13] Internet X.509 public key infrastructure certificate policy and certification practices framework. `http://www.ietf.org/rfc/rfc2527.txt`, 1999.

[14] HTTP over TLS. `http://www.ietf.org/rfc/rfc2818.txt`, 2000.

[15] Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. `http://tools.ietf.org/html/rfc5280`, 2008.

[16] The Secure Sockets Layer (SSL) protocol version 3.0. `http://tools.ietf.org/html/rfc6101`, 2011.

[17] Representation and verification of domain-based application service identity within Internet public key infrastructure using X.509 (PKIX) certificates in the context of Transport Layer Security (TLS). `http://tools.ietf.org/html/rfc6125`, 2011.

[18] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *CRYPTO*, 2009.

[19] Q. Sun, D. Simon, Y.-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted Web browsing traffic. In *S&P*, 2002.

[20] CVE-2009-4831. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4831`, 2009.

[21] J. Viega and M. Messier. *Secure Programming Cookbook for C and C++*. O'Reilly Media, 2007.

[22] N. Vratonjic, J. Freudiger, V. Bindschaedler, and J.-P. Hubaux. The inconvenient truth about Web certificates. In *WEIS*, 2011.

[23] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online – Security analysis of cashier-as-a-service based Web stores. In *S&P*, 2011.