SECURITY SENSOR PROVIDING ANALYSIS OF ENCRYPTED NETWORK DATA

Daniel Hamburg
Integrated Information Systems Group, Ruhr University
Bochum, Germany

York Tüchelmann
Integrated Information Systems Group, Ruhr University
Bochum, Germany

Keywords: Transport Layer Security Sensor, Data Collection, Local Application Proxy, Encrypted Attacks, Intrusion Detection, Intrusion Prevention, Secure Socket Layer.

Abstract: Common Intrusion Detection Systems are susceptible to encrypted attacks, i.e. attacks that employ security protocols to conceal malign data. In this work, we introduce a software sensor, called Transport Layer Security Sensor (TLSS), providing detection engines access to network data encrypted at Transport Layer. Transport Layer Encryption, such as SSL, is typically implemented by a local application and not the OS. TLSS resides on the monitored host and executes cryptographic functions on behalf of local applications. TLSS decrypts incoming encrypted network packets and passes the data to the application, e.g., a Web server software. In addition, cleartext data is also passed to a detection engine for analysis. We present an implementation of TLSS designed for Web servers providing SSL-secured HTTP access and evaluate sensor's performance.

1 INTRODUCTION

Till this day, the transaction volume of online shopping on the Internet grows. One major finding for this growth is that online merchants provided apropriate mechanism to protect transactions (boede + partners integrated marketing, 2004). For instance, a favored technique is the use of Secure Socket Layer (SSL), which emerged to the de-facto standard in protecting web applications. SSL establishes a confidential and integer communication between client and server making wiretapping infeasible. Vice versa, adversaries can deploy these cryptographic secured channels to conceal attacks inside encrypted data. Intrusion Detection Systems (IDS) depend upon the availability of unencrypted data. In particular, Network based Intrusion Detection Systems (NIDS) are blind to attacks encapsulated in encrypted network traffic. Thus, related works propose the use of Host based Intrusion Detection Systems (HIDS) to monitor servers accessible via encrypted tunnels. Though hooking into the operating system (OS), HIDS are blind to attacks inside protocols which encrypt the Transport Layer payload, such as SSL. For instance, web servers and browsers perform cryptographic functions by themselves. The OS passes incoming encrypted SSL data to the corresponding application, which decrypts and processes it. Though, data captured by a HIDS sensor is encrypted.

A standard solution to provide analysis of data encapsulated in encrypted connections is to end the encrypted tunnel at a web proxy, which does not guarantee the non-repudiation assurance. This is crucial in particular for financial transactions. Non repudiation ensures that neither of two business partners is able to dispute his commitment to the transaction afterwards. To bypass this drawback, another common approach decrypts traffic without truncating the tunnel (Breach Security Inc., 2004). For doing so, an intermediary hardware device is equipped with the private key of every original end point. However, export of private keys from the host is a perilous process and should be omitted whenever possible; both, attackers and legitimated insiders (e.g., administrators) which have access to private keys can impersonate the corresponding hosts.

In this work we introduce the *Transport Layer Security Sensor (TLSS)*, which is a special data capturing tool, as part of an IDS. The goal of TLSS is to feature analysis of network traffic encrypted at the Transport Layer, as with SSL. TLSS is a two part software sensor installed on the host to be monitored. Using the sensor it is feasible to capture and block incoming

Hamburg D. and Tüchelmann Y. (2006).
SECURITY SENSOR PROVIDING ANALYSIS OF ENCRYPTED NETWORK DATA.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 172-177
DOI: 10.5220/0001254401720177

network data after it has been decrypted, and outgoing traffic before it will be encrypted. TLSS is capable to forward gathered data to available local (or remote) detection engines. A main benefit is that TLSS neither harms non-repudiation assurances nor does it require export of private keys from the monitored host. Furthermore, we discuss the practicability of TLSS. In a proof-of-concept implementation, we prove evidence that a standard detection engine in combination with TLSS is able to detect attacks encapsulated in encrypted SSL data. In addition, we give a detailed analysis of TLSS's performance.

This paper is structured as follows: In Section 2, we review related works and introduce, elaborate and detail our Transport Layer Security Sensor in Section 3. Afterwards, we describe in Section 4 the proof-of-concept, and summarize and discuss the measured performance results in Section 5. Finally, we conclude our work with a summary of our major results and outlook on future work in Section 6.

2 RELATED WORKS

Ristic (Ristic, 2005) introduces an approach to defend Web servers from malicious data inside encrypted SSL connections. He proposes the integration of an IDS inside the Web server software as loadable module. The basic idea is to hook in the unencrypted web server application's internal information flow. He presents a module for the Apache *httpd* Web server consisting of a proprietary intrusion detection and prevention engine as proof-of-concept. The idea is a straightforward solution for standalone Web servers but prevents integration into existing IDS used to monitor several servers or entire networks. Further, the idea is limited to open source or extendable server software.

Almgren and Lindqvist (Almgren and Lindqvist, 2001) introduce an application-integrated approach to collect data aimed at a server. The authors propose the use of corresponding APIs to hook in the server application, and to take control of data flows inside the application. Gathered data is sent to a detection engine. Further, data flows can be suspended until the detection engine evaluates data, though providing prevention functionality. In addition, the authors provide an implementation for a web server. The basic idea is similar to the one proposed in (Ristic, 2005) but provides cooperation with existing detection engines. Still, the solution is limited to open source server software or software that enables external modules to take over process control. In addition, there is a high development effort, as every monitored application requires a special sensor.

3 TRANSPORT LAYER SECURITY SENSOR (TLSS)

The efficiency of Intrusion Detection Systems highly depends upon the data they analyze, hence encrypted data poses a serious threat. We propose a new sensor, providing detection engines access to data encapsulated in encrypted tunnels. In this context, sensor indicates a data capturing software tool installed on the monitored host, gathering network packets' headers and payload.

3.1 Rationale

Transport Layer Security Protocols, (e.g. SSL, Secure Shell (SSH)), are a favored technique to secure transported data on public networks. Those protocols encrypt Application Layer headers and payload, but do not change headers of protocols on the Internet- or Transport Layer, (e.g., IP and TCP). Cryptographic functions for Transport Layer Security Protocols are generally implemented by an application and not the OS. For instance, common web servers and browsers include cryptographic functions for SSL connections and do not rely on the OS to offer these capabilities. The OS passes incoming encrypted data to the corresponding application which decrypts and processes it. Hence, capturing local data flows between OS and application, e.g. with a HIDS sensor, results in encrypted Application Layer headers and application payload, valueless for common detection engines. Hooking up applications and analyzing unencrypted internal data flows is costly and not feasible for closed source software. Therefore, we propose to outsource the cryptographic functions from the application to a software security module, residing on the same host as the application, and to capture the unencrypted local traffic between this module and the application. Transport Layer Security Sensor (TLSS) integrates both a module providing security functions and a data capturing sensor.

3.2 Design of TLSS

TLSS consists of two parts, the Local Application Proxy (LAP) performing cryptographic functions and a software sensor to capture data. The basic idea of TLSS is to outsource cryptographic functions from the application to LAP, thus, enabling the software sensor to capture unencrypted data exchanged between LAP and application. Incoming encrypted network packets aimed at the application first pass the network stack implemented by the OS. Subsequently, the OS hands the packets to LAP which decrypts and forwards them to the application. On the other hand, the application passes outgoing packets to the LAP

which applies cryptographic mechanisms and hands on secured data to the OS. Local communication between LAP and application is not encrypted, thus we use the software sensor to capture Application Layer headers and payload. In addition, the software sensor captures Internet- and Transport Layer headers from the OS to detect low-level attacks, e.g. illegal combinations of TCP-flags (Daniels and Spafford, 1999).

3.3 Discussion

In contrast to existing solutions, TLSS does not require export of secret keys from the monitored host. In the last resort, private keys are locally migrated from the application to LAP. Typically, keys are stored on the host's hard disc and can be accessed by the local TLSS, rendering migration unnecessary. Use of special cryptographic hardware to store private keys, (e.g. Smart cards), or to accelerate cryptographic computations does not foreclose the use of TLSS. LAP has to be adapted to interact with the hardware using corresponding drivers.

Compared to a web proxy, (also known as *SSL terminator*), TLSS does not affect non-repudiation assurances as it does not truncate secure connections. For instance, SSL allows a client to establish a secured connection to a server, i.e. between the SSL Layers of the two hosts [1]. A web proxy, which is a dedicated host, implements the SSL layer, hence ends the secured connection. The non-repudiation assurance holds between client host and web proxy but not between client and server host. As TLSS resides on the server host, the secure connection terminates at the server, thus ensuring non-repudiation between client and server host.

One limit of TLSS is its boundary to applications that allow outsourcing of cryptographic functions. This does not necessitate open source software, but the possibility to shutdown or bypass the application's internal cryptographic functions. We argue that this assumption holds for most server software used on the Web, as web services usually implement security as optional add on. With TLSS, the server application offers the unsecured service and LAP appends security functions. To do so, the server's OS passes incoming encrypted data, (e.g. according to the TCP-/UDP-port number it is aimed at), to LAP. LAP communicates with the server application either via API function calls or via a virtual local network interface. Using a virtual network interface, TLSS is independent of the server application's API, which is a contrast to application-based IDS, which rely upon the availability of the API. Another advantage is that one TLSS can serve multiple applications. For instance, one LAP can offer SSL-functions to several applications on the same host.

[1] In the TCP/IP network model, the SSL Layer is located between the Transport Layer and the Application Layer

4 PROOF-OF-CONCEPT

We implemented the Transport Layer Security Sensor for a Web server, offering HTTP over SSL access to resources. Local Application Proxy acts as SSL proxy for the server, performing all SSL related functions, (e.g., authentication, key exchange, encryption).

4.1 Implementation

LAP uses the *Stunnel* 4.11 application (Trojnara, 2004) in addition with the *OpenSSL* software libraries (OpenSSL Development Team, 2005) to serve as local SSL proxy for an Apache *httpd* Web server 2.0.52-3.1 (Apache Software Foundation, 2005a) on a Fedora Core 3 Gnu/Linux system.

Figure 1 illustrates the communication between a client and a server equipped with TLSS. In the first phase of communication, client and server establish a SSL connection. Subsequently, the client initiates SSL encrypted HTTP requests via his web browser. The server receives packets, containing requests, on the *eth0* network interface card (NIC). Packets traverse the network stack and are passed to *Stunnel*, which decrypts SSL data and sends the encapsulated HTTP request to *httpd*, using the loopback device *lo*. *lo* is a virtual network interface used for local inter-application communication on Linux systems. Outgoing HTTP responses from *httpd* to the client pass the same way reversely.

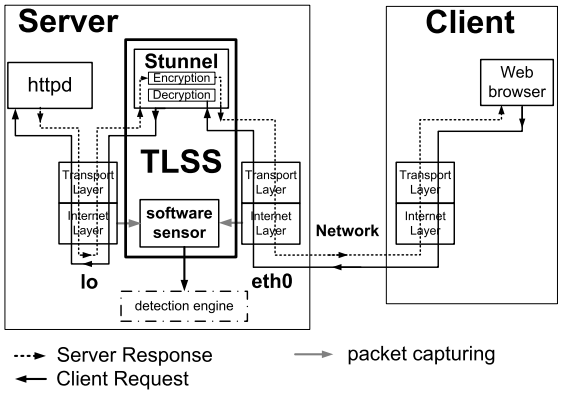

Figure 1: Communication flow between a server equipped with TLSS and a client host.

The software sensor captures both the communication on the *eth0* and the *lo* interface. It captures only IP- and TCP-headers from *eth0* and TCP-payload on

lo, i.e. the decrypted HTTP-headers and -payload. The sensor hands captured packet data to a local detection engine which analyzes packets captured on *eth0* to detect attacks on the Internet and Transport Layer and packets from *lo* to detect attacks inside the encrypted SSL payload, e.g., attacks using hostile HTTP-header and -payload data.

Packets captured on the *lo* interface do not include the correct attacker's IP address and port number, but instead always the IP address 127.0.0.1 in combination with apparent random port numbers. LAP uses those alternative addresses to communicate with the local *httpd* server software. We adapted *Stunnel* to generate a list of mappings:

(Stunnel's port number on loopback interface) $\mapsto$ *(client's IP address, port number)*

The list enables an administrator to link the alternative addresses to the attackers original ones. We plan to adopt LAP to automatically change IP address and port number to the correct values, before sending data to a detection engine.

4.2 Function Test

The focal point of implementation is to demonstrate the functionality of our idea, i.e. making possible the detection of attacks encrypted at Transport Layer. Test cases consist of a client establishing a SSL connection to the server and consequently sending packets with malicious HTTP-requests through the encrypted tunnel. A local *Snort-Inline* IPS (Syngress Author Team, 2004) detection engine on the server host analyzes captured data.

In the first test case, the server is equipped with a TLSS, forwarding traffic captured on the *lo* device to the detection engine. Captured traffic on *lo* is unencrypted, therefore the detection engine traces attacks and generates alarms. TLSS achieves the same results, as well with malicious HTTP-headers, as with malicious HTTP-payload.

The second test case consists of a system equipped with a local sensor but no LAP. The sensor captures packets including TCP payload directly from the NIC, alike HIDS sensors, and passes them to the detection engine. The packet's payload contains malicious data, but as it is nested inside encrypted SSL packets, the detection engine does not observe the attacks.

Results indicate, that TLSS enables IDS to detect attacks inside network traffic encrypted at Transport Layer.

5 PERFORMANCE

TLSS's performance is an important criterion to evaluate the applicability of our idea. Therefore we measured the performance of a Web server equipped with a TLSS. In particular, we want to evaluate LAP's influence on the server's performance, as it executes all SSL functions in lieu of the Web server application. Evaluating software sensor's performance is secondarily, therefore we deactivate data capturing. Performance of data analysis is ignored, as it is not part of TLSS.

As a benchmark, we compare the results to the ones of a reference system which employs the *mod_ssl 2.0.52-3.1* SSL-module.

5.1 Metric and Experimental Setup

The metric *average latency* and *packet loss ratio* indicates system's performance. In this context, latency is defined as the time delay between the moment the client initiates a HTTPS-request and the moment he receives the proper HTTPS-response including all requested objects. Latency is a standard criterion for evaluation of web server performance (Iyengar et al., 1997), while the packet loss ratio indicates server's congestion.

The testbed consists of two HTTPS-server, i.e., the reference system and one equipped with TLSS, three clients [2] and one management station, all interconnected over a switch. Clients, management station and HTTPS-servers have AMD Athlon 64 3000+ CPUs, 512 MByte RAM and Broadcom NetXtreme Ethernet adapters applicable on 100 MBit/s and 1 GBit/s networks. The operating system of all hosts is Fedora Linux Core 3.

Both servers host an identical HTML-page of size 836 Byte, including references to four jpg-pictures [3]. Clients are equipped with the Apache Web testing tool *JMeter* (Apache Software Foundation, 2005b) to simulate Web users' accessing the server, and to obtain values for latency and packet loss ratio. A *JMeter* instance on the management station coordinates the clients.

In a first test case we vary the *encryption algorithm* and the duration of *HTTP Keep Alive*. Encryption algorithm and corresponding key length have direct influence on the performance of SSL sessions (Apostolopoulos et al., 1999), while HTTP Keep Alive influences the HTTP transfer encapsulated in a SSL session, therefore also SSL latency. In addition, we compare results for a 100 MBit/s and 1 GBit/s network. We perform each measurement with 150, 300, 600 and 900 simultaneous user sessions respectively to simulate increasing workload.

[2] In this context, client indicates a host sending HTTPS-requests to the server.

[3] The picture's size is 91.3, 23.5, 14.5 and 3.96 kByte respectively

To do a detailed analysis of system's behavior close to satiation, we employ a second test case, with a 100 MBit/s network connection, no HTTP Keep Alive and 3-DES with 168 Bit key length. The number of parallel user sessions is raised with high granularity (15, 30, 75, 150, 300, 450, 600 parallel sessions). We measure the performance of TLSS with an activated software sensor to evaluate our assumption, that LAP and not the software sensor dominates TLSS's performance. The sensor captures data but does not save it on hard disk.

5.2 Results

Figure 2 represents average latencies of LAP and the reference system using the RC4 encryption algorithm, on both 100 MBit/s and 1 GBit/s network. Latencies for more than 600 user sessions on a 1 GBit/s network are not printed because LAP gets overloaded and starts dropping requests.

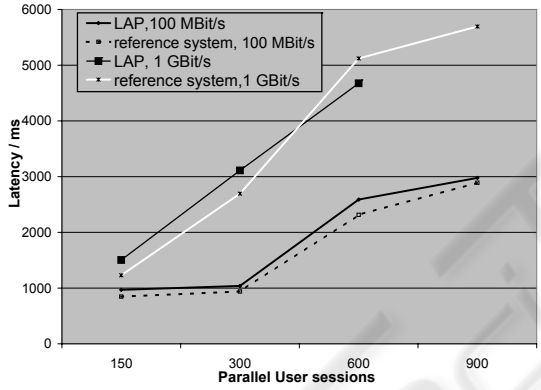

Figure 2: Average Latency for RC4 128 Bit, no HTTP Keep Alive (Test case 1).

It is remarkable, that 100 MBit/s network offers better performance, i.e. minor latency, than 1 GBit/s, both for LAP as for the reference system. One possible reason is that higher number of parallel requests reaches the server on a 1 GBit/s network than on 100 MBit/s. This leads to an increased number of parallel processes initiated by *httpd* [4]. As mentioned in (Iyengar et al., 1997), we assume that, provoked by Web server's internal procedures, the CPU wastes time switching between *httpd* processes which leads to an increased latency .

On a 1 GBit/s network, LAP outperforms the reference system for 600 simultaneous user sessions. This is due to limitations of the Web server software in conjunction with standard settings, allowing a maximum of 256 parallel client sessions. The increased latency of the reference system is caused by requests waiting to be processed by *httpd*. Increasing the number of maximum client sessions results in minor latency.

Figure 3 indicates latencies for AES in combination with varying HTTP Keep Alive Time. Using HTTP Keep Alive decreases the latency for the reference system but not for LAP. We'll discuss this in the next section.

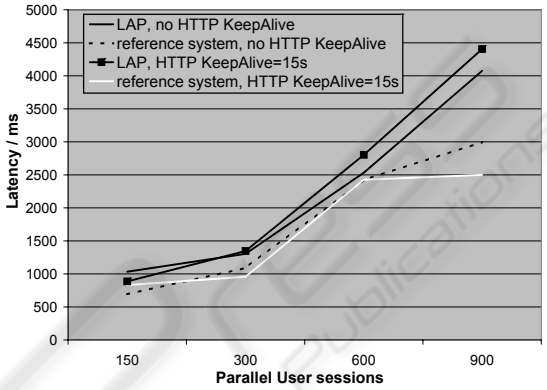

Figure 3: Average Latency for AES 128 Bit, 100 MBit/s (Test case 1).

Figure 4 presents the average latencies of the reference system and TLSS, with activated data capturing.

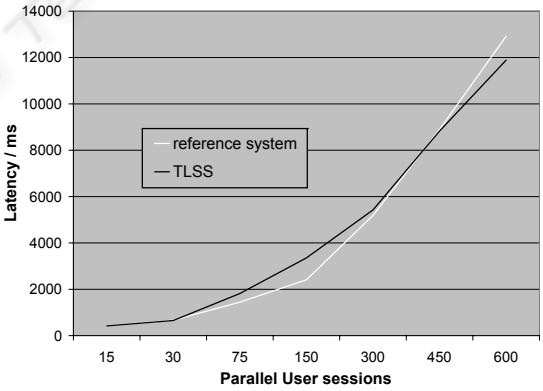

Figure 4: Average Latency for 3DES 168 Bit, no HTTP Keep Alive, 100 MBit/s (Test case 2).

5.3 Analysis of Results

The results indicate that the reference system performs slightly better than LAP, i.e. the reference system has a minor latency. The reference system's performance advantages are due to the fast internal communication between the SSL module, perform-

[4]The Web server is operated in prefork mode, i.e. using one process per client request

ing cryptographic functions, and the Web server software. While the reference system employs API function calls, the communication between LAP and Web server software utilizes the virtual *lo* device. For a small number of parallel sessions, the internal communication takes up most part of the server's computing time, resulting in a high relative performance advantage, (i.e. performance advantage / total latency), for the reference system. For an increasing sessions' quantity, execution of cryptographic algorithms consumes the major part of CPU time. Cryptographic computation's performance [5] is similar for both systems, therefore an increasing number of parallel user sessions diminishes the reference system's relative performance advantage.

As mentioned in the previous section, Figure 3 indicates that activating HTTP Keep Alive does not lead to a decreased latency for LAP. In contrast to LAP, the reference system's SSL module is integrated in the Web server software. Hence, if HTTP Keep Alive is activated on the reference system, the SSL module automatically caches the corresponding SSL sessions which leads to decreased latency. We plan to integrate this feature in an improved version of LAP.

Results presented in Figure 4 indicate that activation of packet capturing has no negative impact on sensor's performance. The maximum capacity handled by LAP is 600 simultaneous user sessions, while reference system handles up to 750 sessions.

Overall results indicate that TLSS slightly decreases the performance of the monitored server and can lead to a decreased availability in case of high workload. This is due to the low performance of the *loopback* interface used for local communication between LAP and server application, because data traverses the network stack of the virtual device. We argue that our implementation of TLSS is suitable for Web servers that do not operate in the range of maximum workload.

6 CONCLUSION AND FUTURE WORKS

In this paper we introduced Transport Layer Security Sensor (TLSS) as part of an Intrusion Detection System. TLSS facilitates detection engines to analyze network traffic encrypted at the Transport Layer, such as SSL. We designed TLSS to cooperate with third-party detection engines, though enabling the integration into existing IDS. In contrast to existing solutions, TLSS neither denies the non-repudiation assurance, nor exports fundamental secrets (e.g. private keys) to other hosts.

Our implementation of TLSS for a SSL-enabled Web server indicates the viability of our idea, but requires further improvements in terms of performance. The focal point is to increase the maximum number of parallel user sessions handled by TLSS.

In addition, we plan to extend TLSS to provide preanalysis and anonymization of data before sending it to a detection engine. While the first will decrease detection engine's workload, latter will prevent that an attacker, wiretapping the communication between engine and sensor, gathers sensitive informations from the transmitted data stream.

[5] Both *Stunnel* and *mod_ssl* use the *OpenSSL* (OpenSSL Development Team, 2005) libraries

REFERENCES

Almgren, M. and Lindqvist, U. (2001). Application-integrated data collection for security monitoring. In *Recent Advances in Intrusion Detection (RAID 2001) Proceedings*, Davis.

Apache Software Foundation, A. (2005a). Apache http server project. http://httpd.apache.org/.

Apache Software Foundation, A. (2005b). Apache jmeter project.

Apostolopoulos, Peris, and Saha (1999). Transport layer security: How much does it really cost? In *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*.

boede + partners integrated marketing (2004). Research - consumer awareness and concerns.

Breach Security Inc., B. (2004). Breach viewl ssl. White Paper.

Daniels, T. and Spafford, E. (1999). Identification of host audit data to detect attacks on low-level ip vulnerabilities. *Journal of Computer Security*, 7(1):3–35.

Iyengar, A., MacNair, E., and Nguyen, T. (1997). An analysis of web server performance. In *Proceedings of GLOBECOM '97*.

OpenSSL Development Team, O. (2005). Openssl project. http://www.openssl.org/.

Ristic, I. (2005). *Apache Security*. O'Reilly Media, Inc.

Syngress Author Team, S. (2004). *Snort 2.1 Intrusion Detection*. Syngress Publishing, Rockland.

Trojnara, M. (2004). Stunnel - ssl encryption wrapper. http://stunnel.mirt.net/.