Safe Configuration of TLS Connections

Beyond Default Settings

Michael Atighetchi, Nathaniel Soule, Partha Pal, Joseph Loyall
Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA 02138
{matighet, nsoule, ppal, jloyall}@bbn.com

Asher Sinclair, Robert Grant
Air Force Research Laboratory
525 Brooks Road, Rome, NY 13441, USA
{asher.sinclair, robert.grant}@af.rl.mil

***Abstract*—Transport Layer Security (TLS) and its precursor Secure Sockets Layer (SSL) are the most widely deployed protocol to establish secure communication over insecure Internet Protocol (IP) networks. Providing a secure session layer on top of TCP, TLS is frequently the first defense layer encountered by adversaries who try to cause loss of confidentiality by sniffing live traffic or loss of integrity using man-in-the-middle attacks. Despite its wide deployment and evolution over the last 18 years, TLS remains vulnerable to a number of threats at the protocol layer and therefore does not provide strong security out-of-the-box, requiring tweaks to its configuration in order to provide the expected security benefits. This paper provides a summary of the current TLS threat surface together with a validated approach for minimizing the risk of TLS-compromise. The main contributions of this paper include 1) identification of configuration options that together maximize security guarantees in the context of recent TLS exploits and 2) specification of expected flows and automated comparison with observed flows to flag inconsistencies.**

Keywords: Transport Layer Security (TLS), Secure Socket Layer (SSL), configuration, secure flow modeling

Distribution A. Approved for public release; distribution unlimited (Case Number 88ABW-2013-2893). This work was sponsored by the Air Force Research Laboratory (AFRL).

I. INTRODUCTION

Ever since its first specification in 1995, Secure Sockets Layer (SSL) has been the predominant way of securing interactions between clients communicating with servers on the Internet. The latest version of this protocol, Transport Layer Security (TLS 1.2) [1], specifies two sub-protocols, a TLS Record Protocol and the TLS Handshake Protocol, aiming to provide confidentiality and integrity of data exchanges between two communicating applications. TLS protects data in transit between two endpoints, with authentication of at least one endpoint to the other. On the Internet, web browsers typically use TLS to validate identities of servers hosting web sites. In Department of Defense (DoD) environments, TLS is frequently used to perform mutual authentication, enabling servers to verify identities of clients and clients to verify identities of servers.

SSL/TLS has evolved over 18 years from SSL 1.0 to TLS 1.2 and has been widely deployed and accepted across Internet servers. This has made it an appealing target for attackers, who continuously launch attacks that directly target the protocol's design. Furthermore, complexities associated with compatibility across a wide range of potential browsers and platforms frequently leave servers with weak configurations that can be exploited by adversaries to cause loss of confidentiality and integrity. Finally, configuring a full set of server endpoints so that they have consistent strong settings is challenging. The difficulty is frequently underestimated and it is easy to miss communication flows, which leads to a lack of coverage. Furthermore, the resulting configurations embody information assurance tradeoffs that are often unclear to the system administrators and users. All of these points together lead to unprotected communications that are assumed to be protected. What makes this even worse is that not only is TLS generally the first layer of defense encountered by adversaries, but often the only one.

The contributions of this paper are two-fold. First, this paper provides a comprehensive summary of the current TLS threat surface together with mitigation strategies and best practice configuration settings for maximizing security guarantees in DoD environments. Second, the paper presents the beginnings of our work on *TLSAnalyzer*, a model-based tool for specifying, observing, and cross checking TLS connections for security properties.

The remainder of the paper is organized as follows. Section II describes related work, Section III introduces the Crumple Zone as an example motivating scenario, and Section IV provides a generic threat model for TLS together with best practice configuration guidance. Section V covers the *TLSAnalyzer* tool. Section VI describes results of using *TLSAnalyzer* during a Red Team exercise, while Section VII describes future work and Section VIII concludes the paper.

II. RELATED WORK

This paper relates to previous work across a variety of categories.

Formal methods: There exists a vast collection of work on applying formal methods to build assurance in security

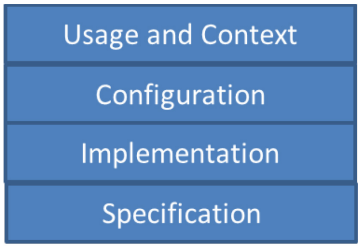

Figure 1. Layered Approach to Verification

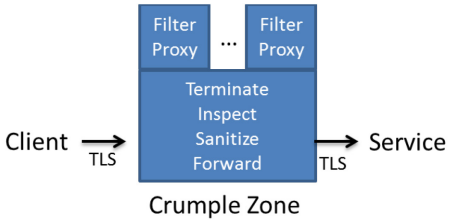

Figure 2. Crumple Zone as a motivating example of TLS configuration analysis

protocols, including TLS. This work operates in large part at the low level protocol layers as seen in the bottom boxes of Figure 1 and seeks to establish a trusted base on which higher levels of validation and verification can occur. The work presented in [2][3] provides a good overview of formal approaches for validating security protocols. Various work related specifically to formal analysis of TLS exists, two such examples include [4] which verifies an F# implementation of TLS 1.0 both symbolically and cryptographic-computationally, and the protocol analysis provided in [5] which evaluates TLS by using the Isabelle theorem-prover to generate proofs over abstract message exchanges. The work described in this paper builds upon and is complementary to formal methods by using empirical information about attacks and engineering techniques to minimize exposure. This paper focuses on securing TLS at the configuration and usage levels under the assumption that the underlying TLS protocol is itself largely secure, albeit with a set of known vulnerabilities.

Automated Configuration Management: The Secure Content Automation Protocol (SCAP) [6] NIST standard specifies a multi-purpose framework for automated configuration, vulnerability and patch checking, technical control compliance, and security measurement. SCAP expresses security assertions in concrete terms and focuses on the configuration state of devices, while the constraints presented in this paper are more abstract and include interactions between multiple components.

Domain Specific Languages (DSLs): Lobster [7] is a DSL for security policy configuration that allows modeling of network flows between security domains. While Lobster provides means for analyzing network models and refining them into enforceable SELinux and Xen policies, it does not provide capabilities to perform cross checking between modeled and observed flows. ConfigChecker [8] uses binary decision diagrams to model the network policy of a set of network devices and uses model checking to perform reachability analysis in order to check end-to-end security properties. Compared to our work, ConfigChecker focuses solely on security analysis and provides no direct means for constructing the models from observed flow information or generating enforceable target representations.

Trustworthy Internet Movement (TIM): The TIM [9] is a non-profit, vendor-neutral organization with the goal of fostering actionable change in IT security practices. The global dashboard generated by SSL Pulse uses an empirical approach similar to the work presented in this paper and provides a large-scale view of TLS properties across Internet web sites. The guidance provided in [10] describes best practices for SSL/TLS with an intended deployment target on the openly available Internet. This paper continues in a similar vein, extended for and focused on controlled DoD environments that have limited backward compatibility and openness constraints.

III. Example Scenario: The Crumple Zone

Service Oriented Architecture (SOA) is a software engineering technology that is increasingly used in many important military and civilian systems. The features that make SOA appealing, like loose coupling, dynamism and composition-oriented system construction, make securing SOA systems complicated. These features ease system development, but introduce additional vulnerabilities and points of entry beyond those that exist in self-contained, static, or stove-piped systems.

To address the security risks found in SOA systems, we have developed a new architectural construct called a crumple zone [11], designed to improve the resilience and survival of SOA services against cyber-attack. A crumple zone is analogous to the crumple zone in an automobile and forms a protective layer that absorbs the effects of attacks by localizing or eliminating the damage they cause and leaving critical components unaffected.

The crumple zone (CZ), shown in Figure 2, is a layer of intelligent service proxies that work together to present a high barrier of entry to the adversary, increase the chance of detection of malicious activities, and contain and recover from failures and undesired conditions caused by malicious attacks. These proxies collectively implement the service's consumer-facing interface. A proxy works by applying security checks and controls on intercepted data, including partial execution of the intercepted data, and approves data release only if those checks pass. Only data that has been inspected and approved by the proxies is passed along to the service. In the process, malicious content and malicious behavior are contained within the CZ. Because the CZ inspects and processes potentially malicious and untrusted data, CZ components are expected to fail occasionally. Therefore, CZ components are monitored by watchdog processes that restart CZ components as necessary.

To be effective, all client interactions with the protected service must be intercepted and routed through the CZ, which establishes a constraint over network flows. For this purpose, the CZ acts as the endpoint for inbound TLS connections initiated by clients, performs the checking

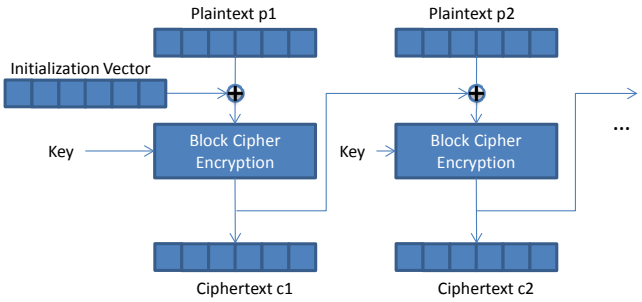

Figure 3. Cipher-block Chaining Vulnerabilities

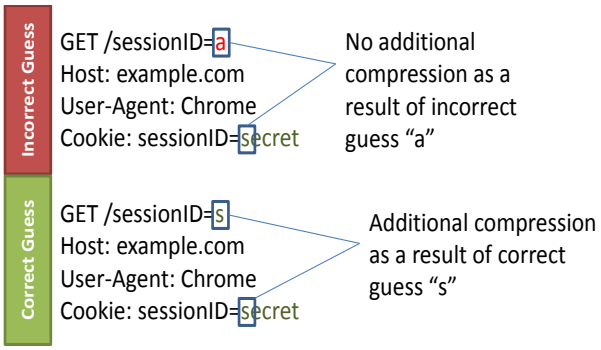

Figure 4. High Level Example of CRIME Attack

functionality, and finally establishes outbound TLS connections to pass the client requests on to the protected service. In this context, it is important to establish the following properties

- Consistency: All flows going in and out of the CZ are protected via TLS
- Non-bypassability: No flows exist that are not inspected by the CZ
- Security: The TLS configuration used provides strong confidentiality and integrity guarantees

IV. TLS Threat Model and Best Practice Configuration Recommendations

A. *TLS Attack Surface*

To ground safe configuration of TLS in reality, it is important to first discuss the associated threat model. The variety of available SSL/TLS versions and implementations coupled with a complex set of configurable parameters leads to an environment where setting up TLS is easy, but setting up TLS correctly and securely is significantly more difficult. This abundance of combinations in the tuple space of <version, implementation, configuration> leads to an expansive attack surface. In recent years numerous attacks on even the latest versions of TLS have been successfully demonstrated and observed in the wild.

Due to the slow and piecewise upgrades of browsers and web applications, a diverse assortment of versions of TLS and its predecessor SSL are in active use across the Internet landscape. To accommodate this diversity, the TLS handshake, which occurs at the start of all TLS connections, enables two potentially disparate versions to agree on a common protocol. This down-negotiation of protocol version means that even an implementation of the latest TLS protocol (1.2 at the time of this writing) may end up communicating with an earlier and less secure protocol. Without proper configuration to restrict this functionality appropriately, a deployed TLS implementation may include the vulnerabilities of all predecessor versions.

Take as an example, the Browser Exploit Against SSL/TLS (BEAST) attack [12]. This attack, whose theoretical feasibility was first discovered in 2002 [13] and which was demonstrated as practical in 2011, exploits a Cipher Block Chaining (CBC) vulnerability present in TLS 1.0 that can allow an attacker to perform a plaintext recovery. In CBC mode, each new block of text to encrypt is first XORed with a nonce. Encryption of the first block (as shown on the left of Figure 3) uses a random initialization vector as a nonce, but encryption of any subsequent block uses the ciphertext of its preceding block as the nonce (as shown on the right of the figure). An attacker can capture the ciphertext for blocks c1 and c2, with c1 being used as the nonce for c2. Next, the attacker proceeds to XOR c1 with a guess for the plaintext P of c2, resulting in Y. Finally, the attacker manipulates the client to use Y as a plaintext. If the attacker guesses correctly, the resulting ciphertext will be identical to c2, enabling the attacker to know that he/she guessed correctly. As of this writing 65.2% of the sites surveyed by SSL-Pulse [14], a global monitoring service, are susceptible to the BEAST attack.

The Compression Ratio Info-leak Made Easy (CRIME) [15] attack discovered in 2012 relies on information leakage from observations of compression behavior to allow attackers to break SSL/TLS encryption. Unlike BEAST, CRIME is not associated with a particular version of SSL or TLS, but instead with a particular feature. Any TLS implementation which is configured to use compression [16] (streaming or block based), such as Deflate [17] based compression, or the SPDY [18] protocol, are susceptible. In this attack requests are made to a server by varying part of the message in an attempt to match an unknown secret within the message (a session ID for example). As seen in Figure 4 when the guess is correct the compression algorithm will be able to use the redundancy introduced by the equality of the guess and the secret to shorten the request. Thus when the message size is shortened (in comparison to an incorrect guess), the attacker knows they have made another correct step in determining the secret. As of this writing 24.1% of all sites surveyed by SSL-Pulse are susceptible to CRIME [14].

TLS allows connection parameters to be renegotiated after the initial handshake is complete. These renegotiations are performed under the protections offered by the existing TLS-enabled connection (i.e. any secondary handshakes occur over the encrypted channel). Despite this, in 2009 a vulnerability [19] in the renegotiation phase of the protocol was discovered that can allow an attacker to inject content into the start of a victim's channel such that the receiving

TLS implementation will accept that content as if it came from the legitimate client. This attack does not impact the confidentiality of the connection, but instead its integrity, as the attacker is unable to decrypt communications. The fact that confidentiality remains intact notwithstanding, the ability to prepend data to victim communications leads to a whole host of exploit possibilities.

In February of 2013, a cryptographic timing attack referred to as Lucky Thirteen [20] was reported. Lucky Thirteen is a form of a padding oracle attack [21], targeted at breaking confidentiality, that introduces new components into an attack described in 2002 by Serge Vaudenay [22]. The original attack has been mostly mitigated, however the new variation has not. In its current form Lucky Thirteen represents a mostly impractical vulnerability for most TLS scenarios as each of the thousands or millions of steps of the attack that are required to decrypt each byte result in connection termination. This implies that a context is required in which TLS connection failures are not reacted to (from a security perspective), and that messages containing the same content in the same location in part of the stream are executed repeatedly. While this reduces the likelihood of widespread general use of this attack, there are still scenarios that meet these conditions (polling via the Simple Mail Transfer Protocol (SMTP) for example), and extensions to this attack are being actively researched.

B. TLS Configuration Best Practice

TLS incorporates a set of swappable lower level technologies and protocols, meaning that not only is the TLS protocol itself a potential source of vulnerability, but the various implementations sitting below it are as well. For a TLS flow to be secure the cipher suite, compression protocol, certificates, key lengths, and certificate authorities must all be secure, and the implementations must all be patched and up to date. In addition to the core required parameters, optional features, such as mutual authentication mean that no out-of-the-box default configuration is likely to be widely appropriate across deployments. Given this complexity, it is not surprising to find that 70% of existing TLS configurations present at least some form of insecure configuration [23]. As of May 2006, the SSL-Pulse survey finds that only 22.6% of the 171,507 web sites surveyed are considered secure [14] by its definition.

Insecurity leading to the statistics described above is due predominantly to inappropriate configuration settings. In order to safely configure TLS connections, one should consider the following list of TLS configuration best practices that when employed offer protections against the known TLS attacks, each of which are described in more detail below:

1. Configuration restriction
2. Mutual authentication
3. Hostname verification

Undertaking these configuration practices when coupled with the model-based validation described in Section V, provides a level of assurance that any given deployment is positioned in a strong security posture with respect to this line of defense.

The configuration described here is targeted for scenarios where security is critical (i.e. the cost of insecurity is greater than the cost of lack of wider availability), or the administrator is in control of both client and server configuration. Requiring all clients to use TLS version 1.2, for example, is a preferred choice for DoD enterprise environments, but may not be an acceptable business decision for services that need to accept the widest array of client devices and software possible.

The configuration best practices are laid out into three pillars: Configuration Restriction, Mutual Authentication, and Hostname Verification.

1) Configuration Restriction

The plethora of configuration parameters available in TLS, over 40 ciphers that ship with Java 7 and tens of options in the providers, some of whose final values depend on external runtime inputs, leave the door open to many of the discussed vulnerabilities. Pluggable protocols and cipher suites, as well as optional features fall into this camp. The configuration restrictions described below serve to curtail the unpredictability and known vulnerabilities in TLS.

Protocol Restriction: During the TLS handshake, the client sends a message indicating the highest protocol version that it is able to support. The server then responds with a message indicating the chosen version to use. This negotiation could be used by a malicious client to cause a server to operate in a less protected mode than it is capable of. An attacker in possession of an exploit for TLS v1.0 such as BEAST could, for example, configure their client to report a maximum capable version of 1.0, triggering the server to use that version (when it may in fact be a 1.2 capable implementation). To protect against this form of attack, TLS allows restricting which versions it can make use of. The best practice, when client capabilities allow, is to restrict to *only* the highest version available: currently TLS v1.2.

Cipher Restriction: As with negotiable protocol versioning, the TLS handshake allows the client to specify the list of cipher suites it is capable of employing. Certain ciphers such as the RC4 stream cipher [24] and the CBC suites when used with TLS 1.0 contain known weaknesses, e.g., easily observable initialization vectors. In addition, the level of integrity provided by TLS may vary across cipher suites. While most practical implementations do, the cipher specification must employ a hash algorithm to provide integrity assurances. The hash algorithm itself is another potential source of vulnerability: use of MD5, for example, is discouraged as known collision attacks exist [25]. Cipher weaknesses could allow a malicious client to mount an attack (such as BEAST) by limiting the list of available cipher suites it provides to a server. To protect against this type of attack, TLS can be configured to only allow a small set of ciphers that are not known to be vulnerable to current

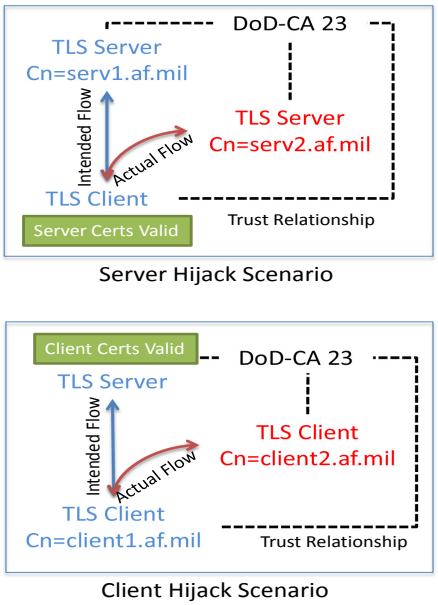

Figure 5. Trusted Certificate Hijack Scenario

exploits. Using the AES256 cipher suite with CBC or Galois Counter Mode (CGM) avoids the increasing list of RC4 weaknesses and is the currently recommended choice.

Options Restriction: The CRIME attack demonstrated a weakness in any available compression technique used with TLS. To protect against this form of attack, compression must be disabled completely.

2) Mutual Authentication

TLS, in addition to providing confidentiality and message integrity (when used with appropriate ciphers), is often used to verify authenticity of endpoints participating in the connection. On the open Internet authenticating servers to clients is commonly the only choice possible, as Certificate Authorities generally do not issue client certificates and companies have mostly relied on company specific password schemes for credentials.

This is quite different for DoD environments, where smart cards such as the Common Access Card (CAC) [26] allow personnel to authenticate themselves to servers via signatures generated using PKI key material on the card. Despite the broad availability and use of CACs, e.g., to log into the Windows operating system, DoD web applications frequently only support password authentication of clients.

For a truly secure conversation to occur, both parties should establish crypto-strong trust in the identity of the other. TLS supports this by a configuration option requiring mutual authentication, forcing both the client and server to authenticate as part of connection establishment. DoD web applications should not only utilize this capability but also follow proper certificate validation procedures to check that certificates presented by clients are properly signed by trusted CAs, have not been revoked, and have not expired. For example, DISA is hosting an enterprise service called Robust Certificate Validation Service (RCVS) that can be accessed using the Online Certificate Status Protocol (OCSP) [27] to validate presented certificates on unclassified government networks.

3) Hostname Verification

While the HTTPS specification [28] requires constraints checking on presented certificates through a process called "endpoint identification", TLS on its own does not. Given certain configurations this could allow a client to open a connection intended for server X, have this server present the certificate for entity Y, and if the client accepts certificate Y, then it will continue on in the connection unaware. There is an implicit assumption in this scenario that the certificate presented, if trusted, matches the one that was expected. This implies that a trusted entity could potentially masquerade as another trusted entity (see Figure 5). While this may sound relatively benign, as both certificates are trusted, this means that compromise of any certificate may make others vulnerable – greatly increasing the attack surface for any service.

The following best practice recommendations can mitigate this attack. For mutually-authenticated HTTPS connections, enable server hostname verification on the client (through configuration) and add explicit code on the server to perform client endpoint identification. Explicit code is needed for the latter case because the HTTPS spec does not explicitly require client endpoint identification. For configurations that run a custom protocol over TLS, we recommend adding hostname verification code on top of the TLS connection code for both clients and servers.

Configuration Example: Java

Another layer of complexity exists in configuring TLS due to the fact that each language, and often each network library on top of that language, will have its own interface (file based, programmatic, or other) for performing this setup. The following paragraphs briefly describe some of the highlights of TLS configuration as it pertains to the Java ecosystem – though similar facilities exist across most modern languages such as Python, and C/C++.

Out of the box all recent Java versions provide the Java Secure Socket Extension (JSSE), which delivers SSL and TLS implementations via a pluggable API. Java 7 introduced support for TLS version 1.2, allowing for protection against vulnerabilities such as BEAST. Recent Java updates to versions 4, 5, 6, and 7 include support for RFC 5746 [29] which addresses the renegotiation vulnerability.

Two of Java's core networking classes, *Socket* and *ServerSocket*, have corresponding *SSLSocket* and *SSLServerSocket* versions. These classes provide methods for restricting the allowed cipher suites and protocol versions. This should be used with the most recent, least vulnerable, and most restricted options available, such as:

- *setEnabledCipherSuites*(new String[] {"TLS_DHE_RSA_WITH_AES_256_CBC_SHA","

TLS_DHE_DSS_WITH_AES_256_CBC_SHA","TLS_RSA_WITH_AES_256_CBC_SHA")

- *setEnabledProtocols(new String[] {* "TLSv1.2"})

The NSA Suite B standard [30][31] , which recommends publically available algorithms for use in classified national security systems, suggests a specific collection based on Elliptic Curve cryptography and AES for protection of sensitive information within the DoD. An implementation of this is expected to become available in 2014 as part of the OpenJDK Java Runtime Environment (JRE) Version 8. Suite B implementations are commercially available through vendors such as IBM. To set the other important properties pertaining to renegotiation, JSSE should be configured in Strict mode by setting the following properties

- *sun.security.ssl.allowUnsafeRenegotiation=false* and
- *sun.security.ssl.allowLegacyHelloMessages=false*.

Since none of the standard JSSE TLS providers offer compression functionality, no specific attention needs to be paid to disable it. However, application level compression might still be at play and needs to be carefully controlled.

Unfortunately, in many cases direct access to the socket level code is not possible, such as when using wrapper libraries for higher level protocols or when making use of application servers. Each of these, as well as other core Java packages such as the asynchronous IO (nio) libraries, provide their own mechanism for specifying TLS properties – though setting system properties will often work across libraries.

V. THE TLSANALYZER TOOL

While the TLS configurations listed in the previous section provide a structured way to define network flow policy, there is no guarantee that the configurations are actually enforced as specified. For JSSE, it is quite common for a one letter typo in a system wide property to cause the TLS connection management to revert back to default mode, without software developers noticing. Another common mistake is to only protect some socket factories with TLS, leaving others unintentionally unprotected.

To address these concerns, we created the *TLSAnalyzer* tool for experimentally constructing network flow models based on observed network traffic and analyzing the resulting models. For constructing an observed model of network flows, *TLSAnalyzer* uses the *tshark* [32] packet sniffer plugged into the network at strategic locations. In the CZ example, *TLSAnalyzer* runs on the CZ machine as displayed in Figure 6 and is configured to report traffic on the following networks:

- The external network between clients and the CZ
- The internal net between the CZ and the protected service
- The management network used to control CZ components
- The loopback network used for local-only communication between intra-CZ components

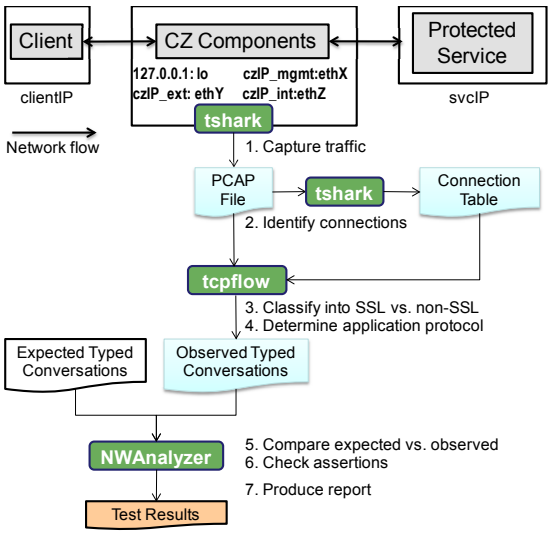

Figure 6. Model Extraction Setup for Network Flows

When deploying TLSAnalzer in larger enterprise environments, it is important to deploy monitoring probes in strategic locations that maximize coverage over raw observables, and there are interesting questions about the recommended probe deployments that are currently beyond the scope of this paper.

After starting the network sniffer, it is important to exercise the system under test by running all available automated tests, to ensure coverage over logic that creates network connections on demand. The output of this first step is a packet capture file (PCAP) containing raw packet content together with timestamps.

In the second step, *TLSAnalyzer* invokes *tshark* to create a connection table that captures all observed TCP flows. Note that using the functionality directly provided by *tshark* (-z conv,tcp) is not sufficient for this purpose, as source and destination addresses are not correlated to establishment of the TCP connection via SYN requests. To achieve the correct semantics, the *TLSAnalyzer* calls *tshark* with a custom set of parameters as follows:

```
tshark -r <pcapfile> -R "tcp.flags.syn == 1
&& tcp.flags.ack == 0" -T fields -e ip.src -
e tcp.srcport -e ip.dst -e tcp.dstport
```

In the third step, *TLSAnalyzer* iterates over the connection table and uses the *tcpflow* [33] utility to reassemble the application stream from multiple TCP packets. Next, it classifies traffic into TLS vs. non-TLS by running regular expression checks over the application stream, i.e., looking for "Client Hello" and "Encrypted Handshake Message" TLS messages. *TLSAnalyzer* also identifies non-TLS traffic (step 4) based on regular expression matches for RMI streams, serialized Java objects, SOAP protocol messages, and HTTP protocol messages, none of which would be detectable in an encrypted stream.

As shown in Figure 6, the *tcpflow*-based processing

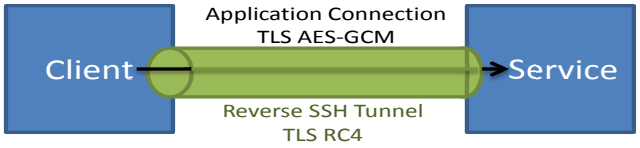

Figure 7. Combining Multiple Ciphers for Extended Protection

produces a list of "observed" enforcement obligations. Step 5 then compares this list with a set of expected communications, as manually specified for a domain expert. The differencing algorithms implemented by the *NWAnalyzer* component can spot a number of misconfigurations, including

- Flows that are expected but not observed
- Flows that observed but not expected
- Flows that are expected to be TLS but are not

In the context of the CZ example, step 6 involves running a customizable set of assertions against the communication model, including the following:

- **Non-bypassibility**: No direct connections between clientIP and svcIP.
- **Protection consistency**: All connections between clientIP and czIP_ext must be TLS.
- **Binding specificity**: All connections related to internal CZ communication identified by port (e.g., log analysis, key sharing and splitting control, firewall actuation) have 127.0.0.1 as source and target IP.

The final step of *TLSAnalyzer* involves generating reports that can be used by subject matter experts to adjust either (1) the implementation and rerun the tools until all constraints are fulfilled or (2) the assertions or description of expected conversations to line up expected constraints with implementation reality.

VI. Experimental Validation

The *TLSAnalyzer* tool described in this paper was evaluated as part of a Red Team Exercise [34] performed by an independent US Air Force Research Laboratory (AFRL) Red Team to assess the efficacy of the Crumple Zone.

One of the experiments conducted during the exercise was to find a bypass to the crumple zone, either by communicating with services that do not perform crypto-strong authentication (via TLS) or by virtue of directly connecting to the protected service. The *TLSAnalyzer* tool was used during both normal operations and attack scenarios executed against the CZ, comparing expected with observed connection patterns.

The following lines show example entries from the connection table that *TLSAnalyzer* generated:

```
192.168.7.119,35861,192.168.7.170,3873->
     plain,Serializable,rmi

192.168.1.68,43813,192.168.6.119,2222 ->
     ssl,na
```

Figure 8. Increase Availability through Redundant Diversity

The first line shows that *TLSAnalyzer* determined a flow to be a TCP connection (plain) on which Serializable Java objects are seen. The second line shows an SSL connection with an unknown application level protocols running over the connection.

Performance and real-time execution was not an issue in the current validation context, as the tool only had to deal with less than 100 connection entries.

VII. Future Work

While a model-based differencing approach in general and the *TLSAnalyzer* tool specifically were found to be useful during development and red team evaluation, there are a number of ways they can be extended to provide enhanced configuration management.

TLS parameter checking: Currently, *TLSAnalyzer* only performs a binary classification of TCP connections as either plain text or protected via SSL/TLS. One clear extension is to extend the tool to include checks for specification attributes of the TLS connection as it gets negotiated. This will involve inclusion of active probing tools, such as SSLScan [35], to test whether servers can be down-negotiated or not. The results can be cross checked against specific constraints in the connection table on a per flow basis or through global assertions, e.g., no connection should be down-negotiable.

Dual-Layer TLS: The configuration practices described above are focused on scenarios where both server and client are free to select among any existing protocol version. Even outside of the consumer browser market there are situations where this is not the case. One approach to achieving strong crypto protection is to combine multiple weak algorithms in strategic ways. Figure 7 shows an example of combining a RC4 stream cipher used in a reverse Secure Shell (SSH) tunnel with a AES-GCM block chaining cipher used for application-level connections that connect through the tunnel. This makes attacks targeting confidentiality and integrity very difficult to execute, as the attacks need to work through two diverse algorithms to succeed. To increase availability, a service can run two tunnels, e.g., one using the RC4 stream cipher and another one using the AES-GCM block cipher, as shown in Figure 8. Exploits aimed at getting ciphers out of sync will likely only succeed against one of the two tunnels.

These scenarios suggest new requirements for *TLSAnalyzer* to support going forward. Two new analyses would be required to provide confidence that a dual layer TLS implementation was configured and operating as desired. The first analysis checks constraints of a single

tunnel, namely ensuring that certain properties are the same (e.g., key length) and certain properties must be different (e.g., cipher types as in stream vs. block). The second analysis checks constraints across multiple tunnels.

The final aspect of future work involves a more extended validation of the tool, including measurements of scalability, latency, and ease of use.

VIII. CONCLUSION

As interactions in distributed systems are protected using TLS, it is critical to validate that the security properties of the resulting network flows are indeed as expected. This paper describes a validated approach, embodied in the *TLSAnalyzer* tool, enabling construction of assurance arguments for network flow policies in distributed composed systems. Furthermore, the paper outlines a number of currently relevant threats against TLS and describes configuration settings aimed at minimizing the attack surface in tightly controlled DoD environments. Going forward, there is opportunity to refine *TLSAnalyzer* as it is adopted by a larger community and enhanced to support more intricate dual-layer encryption scenarios.

ACKNOWLEDGMENT

The authors acknowledge Charles Payne from Adventium Labs for his help in the writing of this paper.

REFERENCES

[1] T. Dierks, E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, August 2008

[2] M. Catherine, "Formal methods for cryptographic protocol analysis: Emerging issues and trends.", Selected Areas in Communications, IEEE Journal, vol. 21, num. 1, pp. 44-54, 2003.

[3] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe, Modelling and Analysis of Security Protocols, Addison-Wesley, 2001

[4] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Cryptographically verified implementations for TLS", Proceedings of the 15th ACM conference on Computer and communications security, 2008

[5] L. Paulson, "Inductive analysis of the Internet protocol TLS", ACM Trans. Inf. Syst. Secur., *vol.* 2, num. 3, pp. 332-351, August 1999.

[6] NIST, (2012). *Security Content Automation Protocol* [Online] . Available: http://scap.nist.gov/

[7] J. Hurd, et al, "Lobster: A domain specific language for selinux policies", Galois internal report, 2008.

[8] E. Al-Shaeret, et al, "Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security", IEEE International Conference in Network Protocols (ICNP' 09), October 2009.

[9] Trustworthy Internet Movement. (May 2013). *Trustworthy Internet Movement - About* [Online] . Available: https://www.trustworthyinternet.org/

[10] I. Ristic. (2013) *SSL/TLS Deployment Best Practices* [Online]. Available https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices_1.1.pdf, April 2013

[11] M. Atighetchi, et al, "Crumple Zones: Absorbing Attack Effects Before They Become a Problem," *CrossTalk - The Journal Of Defense Software Engineering*, March/April 2011D. Goodin. (2011). *Hackers break SSL encryption used by millions of sites* [Online]. Available: http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/

[12] P. Rogaway. (2004) . *Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures* [Online]. Available: http://www.openssl.org/~bodo/tls-cbc.txt

[13] Trustworthy Internet Movement (2013) *SSL Pulse - Survey of the SSL Implementation of the Most Popular Web Sites* [Online]. Available: https://www.trustworthyinternet.org/ssl-pulse/, May 2013,

[14] D. Goodin (2012). *Crack in Internet's foundation of trust allows HTTPS session hijacking*. [Online]. Available: http://arstechnica.com/security/2012/09/crime-hijacks-https-sessions/

[15] S. Hollenbeck. (2004). *Transport layer security protocol compression methods* [Online]. Available: http://tools.ietf.org/pdf/rfc3749.pdf

[16] L.P. Deutsch (1996). *DEFLATE compressed data format specification version 1.3*. [Online]. Available: http://tools.ietf.org/pdf/rfc1951.pdf

[17] M. Belshe, and R. Peon. (2012) *SPDY Protocol*. [Online]. Available: http://tools.ietf.org/pdf/draft-mbelshe-httpbis-spdy-00.pdf

[18] T. Zoller. (2011) *TLS/SSLv3 renegotiation vulnerability explained* [Online]. http://www.g-sec.lu/practicaltls.pdf

[19] N. AlFardanJ., and K. G. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. (2013).

[20] J. Rizzo, and T. Duong. *Practical padding oracle attacks*. Proceedings of the 4th USENIX conference on Offensive technologies, WOOT. Vol. 10. 2010.

[21] S. Vaudenay. "Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS..." Advances in Cryptology—EUROCRYPT 2002. Springer Berlin Heidelberg, 2002.

[22] I. Ristic (2011). *State of SSL. Talk at InfoSec World.*

[23] N. AlFardan, D. Bernstein, K. Paterson, B. Poettering, J. Schuldt. (2013). On the Security of RC4 in TLS. Available: http://www.isg.rhul.ac.uk/tls/

[24] Stevens, Marc. "On collisions for MD5." TU Eindhoven MSc thesis, [Online]. Available: http://www. win. tue. nl/hashclash/On% 20Collisions% 20for% 20MD5 (2007).

[25] DoD Common Access Card. (2013). *CAC Homepage* [Online]. Available: http://www.cac.mil/

[26] M. Myers, R. Ankney, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol (OCSP)", IETF RFC 2560, June 1999, http://tools.ietf.org/html/rfc2560

[27] E. Rescorla, *HTTP Over TLS*, *IETF RFC 2818*. [Online]. Available: https://tools.ietf.org/html/rfc2818, May 2000.

[28] E. Rescorla, S. Dispensa, N. Oscov, *Transport Layer Security (TLS) Renegotiation Indication Extension, IETF RFC 5746*. [Online]. Available: http://tools.ietf.org/html/rfc5746, February 2010

[29] NSA. (2013). *NSA Sutie B Cryptography – NSA/CSS*. [Online] http://www.nsa.gov/ia/programs/suiteb_cryptography/, May 2013

[30] M. Salter, R. Housley, *Suite B Profile for Transport Layer Security (TLS), IETC RFC 6460*. [Online]. Available: http://tools.ietf.org/html/rfc6460, January 2012.

[31] Wireshark. (2013). *Wireshark homepage* [Online]. Availble: http://www.wireshark.org/

[32] TcpFlow. (2013) *Tcpflow homepage* [Online]. Available: http://afflib.org/software/tcpflow

[33] P. Pal, M. Atighetchi, A. Gronosky, J. Loyall, C. Payne, A. Sinclair, B. Froberg, and R. Grant, "Cooperative Red Teaming of a Prototype Survivable Service-Oriented System," Military Communications Conference (MILCOM), Orlando, Florida, October 29-November 1, 2012.

[34] SSLScan (2013). *SSLScan – Fast SSL Scanner* [Online]. Available: http://sourceforge.net/p/sslscan.