Trust Through Origin and Integrity: Protection of Client Code for Improved Cloud Security

Anders Fongen, Kirsi Helkala and Mass Soldal Lund
Norwegian Military University College, Cyber Defense Academy
Lillehammer, Norway
Email: anders@fongen.no

***Abstract*—Military computing is migrating to cloud architecture for several reasons, one of them is the opportunities for improved security management. One opportunity is to ensure that cloud clients are running approved and untainted program code, provided as a proof presented to the cloud service. Such proofs can extend the trust in the client's integrity further than what traditional access control protocols can provide. While access control protocols can ensure that a computer is operated by authorized and trained personnel, they cannot ensure that the client computer is unaffected by malware or poor software control. Problems related to illegitimate program code cannot, in general, be solved by traditional security protocols. The contribution of this paper is an arrangement whereby proof of software approval and integrity can be established, exchanged and validated during service invocations. The demonstration program is a chat forum where the exchanged messages are signed and validated in the client computers, a typical use case which may benefit from our contribution. Two different client-server protocols were tested in order to study the applicability of our contribution.**

***Keywords*—cloud security; integrity attestation; trusted computing; Google ChromeOS**

I. INTRODUCTION

Military computing applications are being migrated to cloud architecture due to a number of advantages, including those related to security management [1] [2].

The integrity of client code is important in most cloud application, but of particular importance where sensor readings and cryptographic operations are involved. Cloud computing relies on mutual trust between client and the service, trust in that the transactions between them take place in a *bona fide* manner. The service offers its interface to a client which is presumed to operate through it in a responsible manner. The mutual trust is usually derived from *authentication* of the person who is operating the client computer, together with personnel management procedures that ensure the loyalty and competence of this person.

Authentication does not extend the trust to the software in use, however. Malware, version mismatch, unauthorized modification and updates may cause the interface to be operated in a harmful manner, causing leaked or falsified information and loss of trust in the system. What is needed is a proof of untainted client software which can be verified by the service during the authentication process. For the remainder of this paper this proof will be called an *integrity attest*. Likewise, the service may attest its software integrity to the client, but we are less concerned about the software integrity in a tightly controlled server environment. Military use cases for integrity attests include sensor readings and cryptographic operations, where client malware may modify, leak or spoof information sent to an unsuspicious server.

This paper will discuss and demonstrate schemes for integrity attestation and how they may be combined with personal credentials in trust management operations. A demonstrator application will be briefly presented. The application will employ the properties of ChromeOS together with the *Cross-origin resource sharing* (CORS) protocol and client-authenticated *Transport Layer Security* (TLS) connections to provide the necessary guarantees for browser based client programs written in Javascript. This demonstrator application employs the *Web Cryptography API* (WCA) [3] which also gives useful insight in the cryptography operation and key management in this environment. The novelty of the contribution is a new application of existing security protocols in order to offer protection of client integrity.

A. Desired security properties

The goal for any computer program is that it behaves as expected and conducts its transactions in a "bona-fide" manner. Since this property cannot be assessed in general, we choose to replace it with the following requirement:

"Only approved client code may access a given service"

This requirement entails that software running in the client has been inspected and approved during development, and protected from hostile modifications during deployment and execution. If adequate procedures for development and deployment of client code are in effect, this requirement will be equivalent to the required "bona-fide" operation of the client.

The technology elements taken into regard in this paper for establishing the required trust are:

1) Platform integrity protection
2) Hardware bound keys and certificates
3) The Cross Origin Resource Sharing (CORS) protocol
4) *Indication of Origin* in the HTTP headers
5) Device Security Policy Management
6) Mutually authenticated TLS connections

B. Related work

The protection of software integrity has been a prominent research field for decades. Oldest is likely to be the *process separation* found in any operating system, and the *virus detection* programs that aim to recognize binary fingerprints of well known malware types. *Code signing* is used to authenticate the software creator and to detect any changes to the software since release. The introduction of *Application Stores* in recent

 ISBN: 978-1-61208-821-1

operating systems offers code signing as well as life-cycle management of the distribution, deployment, upgrading and removal of software. These research areas are distantly related to the presented research efforts, but they all fail if the protection mechanisms themselves are attacked. Hardware assisted protection mechanisms are needed, as operating systems designers have known for 50 years.

A combination of hardware assisted integrity protection and identity management is shown in [4], but the solution presented there does not protect software above the platform level. To the authors' best knowledge, no other platform than ChromeOS offer hardware-assisted integrity protection of the entire software stack, and the protection arrangements presented in this paper is believed to be novel.

The remainder of the paper is organized as follows: In Section II a discussion on the technology elements used in the presented protection scheme will be presented, followed by a model for the client code protection in Section III. A proof-of-concept prototype for evaluation of the protection model is presented in Section IV, follows by a conclusion in Section V.

II. Technology discussion

This section provides a more detailed discussion of the technology elements involved in the aforementioned protection scheme.

A. Platform integrity protection

The integrity of the software stack can be protected through inspection techniques, where either (1) patterns of known malware is detected or (2) through detection of any modifications from an approved/correct state through the verification of hash values. For the latter approach, the Trusted Platform Module (TPM) [5] offers a range of services for boot-time software inspection, which also aids the protection of non-volatile storage in case the platform has been compromised.

Other techniques include the verification of a digital signature created over the software image, to verify the integrity of the software as well as its source. This approach is taken by Google ChromeOS [6], where Google's public key and signature verification code is located in ROM and executed during bootstrap. ChromeOS will not operate unless the verification stage has completed successfully, and it cannot be booted from USB memory. Likewise, the ARM processor architecture may use its *TrustZone* mode to establish a verified boot in a stepwise process similar to the TPM [7].

Two limitations are apparent in this arrangement: (1) The platform code is inspected only during bootstrap, and (2) the application programs are not inspected. Limitation no. 1 is due to feasibility reasons. Software inspection must be an atomic operation so task switching and interrupt handling must be disabled for the duration of the operation. It is therefore executed during bootstrap in order not to disturb other activities in the computer. Limitation no. 2 is probably due to the dynamicity and multi-vendor nature of the application programs in use. However, limitation no.2 is also the reason for concern over how malware can enter into the application software and challenge the integrity of the entire platform through exploits of vulnerabilities in its process separation and access control.

Among the well known and current platforms, only ChromeOS verifies the entire software stack, including the applications (which are limited to the Chrome browser, a file manager and a media player). Application code running as Javascript in the web browser is not integrity checked since it is loaded after boot-time, but Section II-C will show how loaded Javascript can be trusted both by its integrity and its origin. For other platforms, device security policy management may offer some protection from malware inside application code (cf. Sect II-E).

B. Hardware bound keys and certificates

Private keys are used to authenticate users or devices. In the former case, the private key should be accessible from all devices operating on behalf of this user. Such private keys can successfully be stored in USB dongles, smart cards, etc. In the latter case, the private key should be bound to the device hardware in a manner where it cannot be exported elsewhere. The typical hardware solution for private key storage is the TPM. For the protection arrangement presented in this paper, the binding of a key (and its certificate) to a device is crucial in order to establish the identity of the device and its associated properties.

Several platforms allow certificates and keys to be installed and bound to the hardware device, e.g., Windows 10, Android and ChromeOS. Certificates can be designed to authenticate both the user and the device, and allow the other party to make assumptions about the identity of the user as well as the properties of the software platform of the device. The presentation of a device-bound certificate during a transaction may (depending on platform) indicate a successful bootstrap integrity control. Within the limitations identified in Section II-A, the combination of integrity control and device bound keys provides *integrity attestation*.

Among the well known platforms, ChromeOS has the most complete assurances, since its bootstrap integrity control also includes the application software: When a client proves the ownership of a user certificate known to be bound to a ChromeOS device, e.g., during establishment of a client-authenticated TLS connection, the service can safely assume that the client device is free of malware and operates as expected (disregarding potential software bugs for the moment). The assumption relies on key management procedures where trusted personnel install the correct keys and certificates in the device during the device deployment phase, and later as certificates expire.

C. The CORS protocol

Inside a browser there are restrictions on where the Javascript code can set up network connections. Originally, there was a *same origin policy* in effect, i.e., connections could only be made using the same scheme, IP address and port as was used to load the web page [8]. Although originally designed to inhibit rogue Javascript programs from leaking information to arbitrary receivers, the restriction also protects the service from access from unauthorized clients; only Javascript loaded from the same server could access the service, which allowed the content of the client code to be

 ISBN: 978-1-61208-821-1

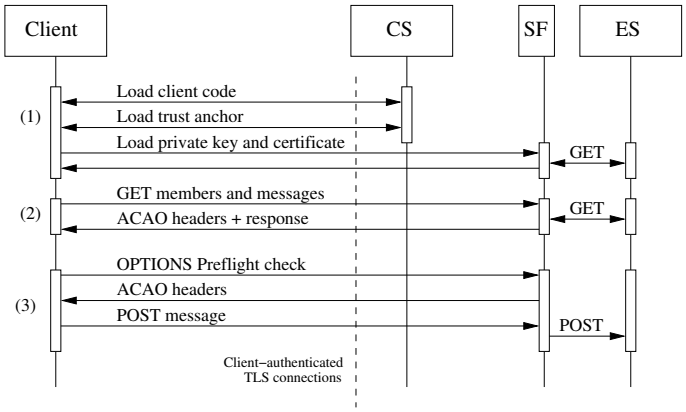

Fig. 1. The protocol elements of the Forum application, CS is the Code Service, ES the Execution Service and SF the Servlet Filter. Details are explained in Section IV.

closely inspected and protected. The use of *Content-Security-Policy* (CSP) directives in the code adds to the robustness of this arrangement [9].

The *same origin policy* has since been relaxed to allow Javascript access to services which explicitly permit connections from designated origins. Termed *Cross Origin Resource Sharing* (CORS), this protocol adds new HTTP headers for the client to request a list of *allowed origins* from the service [10]. These headers are termed *Access-Control-Allow-Origin* (ACAO), as shown in Figure 1. A POST method requires a "preflight-check" in the form of an OPTION method to obtain the ACAO headers prior to the actual POST method, which is not necessary for a GET method. In both cases, the operation is aborted if the ACAO headers do not contain the necessary values.

It is the responsibility of the web browser to enforce these rules, and in order to use the CORS protocol to protect the service from rogue client code, it must be evident that the browser is in fact obeying the CORS rules. Integrity attestation may provide such evidence if the mechanism also verifies this particular property of the browser in use. Only ChromeOS verifies the integrity of the Chrome browser (which is the only browser allowed), while other platforms will need additional measures to obtain the necessary proofs.

D. Indication of Origin HTTP header

As an alternative to the CORS protocol mechanisms, the browser may include a HTTP header to indicate the origin (the URL of the requesting web page, not the IP address of the client computer) of the HTTP request. The service may use this information to complete or abort the operation. The access approval decision takes place in the service, which would need to return an HTTP status code `403 Forbidden` if the origin value indicates an unauthorized source. Error handling in the client is therefore different from CORS use, where the error would be handled in a `catch` block. The use of the Origin header is a fall-back strategy when the client-server communication uses the WebSocket protocol, which does not obey the CORS protocol.

E. Device Security Policy Management

Devices can be subject to mandatory security policy management through installation of software for *Mobile Device Management* (MDM). Allowing only "whitelisted" applications to be installed ensures that only approved web browsers can be used. MDMs are comprehensive frameworks and have not been investigated for the purpose of this paper, but it is possible that MDMs can compensate for the lack of application-level integrity inspection in, e.g., Android.

MDMs can also be used to further reduce the risk coming from disloyal, untrained or careless users, who may change the network configuration to use a different DNS service, install new trusted root certificates, bypass the certification validation during TLS connections, etc. An MDM may enforce a policy that such actions require elevated user privileges. In particular, ChromeOS devices can be subject to *Chrome Device Management* to reduce the risk from these actions [11].

III. Suggested model for protection of client code

The desired property for a client is that it only runs code approved for the given service. This property should be validated by the service itself, which will deny any access from unauthorized code. The presented protection model is targeting browser based client code written in Javascript. The validation relies on a chain of trust elements:

1) During the establishment of a client-authenticated TLS connection to the service, the client presents a certificate that is known to belong to a computer with integrity protection of both platform and browser. In the presented implementation, specific values in the Distinguished Name *OU* element are used to indicate this property.
2) A carefully implemented device administration procedure is in effect to ensure that correct certificates are bound to the respective hardware devices, cf. Section II-B.
3) An uncompromised operating system and web browser will obey the CORS protocol rules, and the service will know that Javascript calls only come from approved software. Alternatively, the `Origin:` HTTP header value may be trusted to determine the source of the client code. By system management procedures, the approved software source will be trusted to contain only well inspected and verified code.
4) Javascript program code is always loaded through TLS (HTTPS) connections, which protect the integrity of the code during transport.

This chain of trust does not become stronger than its weakest link, so a number of reservations apply:

1) The CORS protocol/Indication of Origin relies on correct IP address values from the DNS service. The DNS service never authenticates itself to its clients, and the IP address of the DNS service can be forged, e.g., through manipulation of the DHCP (Dynamic Host Configuration Protocol) service, or by overriding the network configuration on the client

computer. If the connection is TLS protected, a falsified certificate would also be needed for a successful CORS attack, see no.3 below.

2) There are several known attacks on the TLS protocol, as summarized in RFC7457 [12] as well as the more recent Heartbleed and Robot attacks.
3) There is an excessive number of trusted root certificates in the default configuration of the main web browsers. If any of these roots are compromised, they may sign fake certificates that will be validated by the browser and jeopardize the authentication operation in either direction.
4) The standard configuration of a browser allows the user to override an unsuccessful certificate validation during a TLS connection establishment (although trusted not to), so the connection may be completed despite the invalid certificate.

IV. Experimental evaluation of the model

For demonstration purposes, and for a detailed investigation on the feasibility of the presented model, a message chat forum application was programmed. Figure 2 contains a screen shot from the Forum application. The application requirements were as follows:

- All clients fetch their client code from the *Code Service* (abbreviated CS).
- All clients connect to the *Execution Service* (ES) for services using client-authenticated TLS.
- The browsers need to install keys and certificates issued by one specific Certificate Authority.
- A client posts messages with a digital signature. They will be received by all other connected clients.
- Received messages will be validated for correct signature and a valid certificate, and given a trust rating.
- Received messages will be listed on the user interface.
- A list of the names of connected clients will be shown on the user interface.

Digital signatures on messages were created and validated for the reason to explore end-to-end security mechanisms in Javascript. The *Web Cryptography API* [3] was used and provided the authors with useful experience with WCA and related libraries for key management and cryptographic operations.

Figure 1 shows the protocol elements of the Forum application in three blocks: (1) is executed as the client program starts, (2) is executed at regular intervals as a polling operation, (3) is executed each time the user sends a message to the forum. Since the client code is not loaded from ES (Execution Service), all accesses to ES must obey the CORS rules. During a GET operation, the ACAO headers are returned with the returned value (block 2), whereas a POST operation requires a preflight check as shown in block 3. Note how the Servlet Filter (SF) handles the ACAO headers isolated from the application code in ES.

Two alternative implementations of the applications were programmed, based on HTTP and WebSocket protocols, respectively. (1) The use of HTTP protocol allows for a straightforward Servlet implementation on the service side and invoking the `XMLHttpRequest` object on the client side. The CORS protocol will be used together with the inspection of the TLS client certificate to enforce the desired access policy. (2) The WebSocket protocol does not employ the CORS protocol, so inspection of the `Origin:` HTTP header value will serve as a replacement. The TLS client side certificate will still need to be inspected by the service. Details related to the two implementations are described in Sections IV-B and IV-C, respectively.

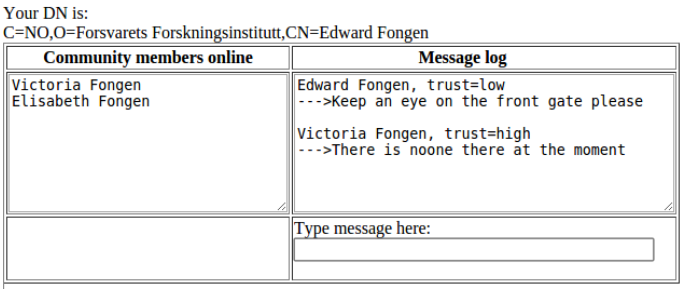

Fig. 2. A screen shot from the chat forum application. Messages from "Victoria" are sent from a ChromeOS device and are marked `trust=high`. Messages from "Edward" are sent from a MacBook.

A. Loading of keys and certificates

The Javascript environment does not have access to the browser's keystore, so the necessary public and private keys will need to be loaded from elsewhere. The chosen solution was to load the private key and certificate from the Execution Service (ES) over a client-authenticated TLS connection. ES returns the private key corresponding to the certificate used for TLS authentication, effectively copying the keys from the browser's keystore to the Javascript environment, where they are imported into the WCA for subsequent signature and validation operation. This solution appeared to be the best of only poor options, but a private key needs better protection than this. The trust anchor is loaded from the Code Service (CS) and also imported into WCA.

The service (ES) does not validate any signatures on the message traffic, and does not need any other keys than what is necessary for the TLS protocol. Signatures are validated by the clients, while ES is merely relaying messages.

B. Managing CORS protocol and client certificate

The service program was implemented as a Java Servlet, and during a client-authenticated TLS connection the client certificate is made available to the invoked servlet through the `HttpServletRequest` object. This certificate may be inspected for any access control purposes, i.e., to allow only clients with attested integrity to invoke services. Although the Java Servlet interface does handle HTTP OPTION methods, the processing of the CORS "preflight checks" is done by a Servlet Filter for reasons of loose coupling between protocol handling and application logic. The Servlet Filter manages both the certificate inspection as well as the CORS protocol, decoupled from the Servlet application logic. The certificate

 ISBN: 978-1-61208-821-1

inspection only accepts one specific issuing CA. Parts of the code in the Servlet Filter is shown in Figure 3.

Being able to inspect the client TLS certificate is essential for the protection arrangement based on attested integrity. Java Servlets (and PHP to some extent) appears to be the only widespread server technologies to offer this opportunity.

C. Inspection of the Origin header in WebSocket communication

The WebSocket protocol is an asynchronous communication protocol, which allows for push-based information exchange. Push-based dissemination offers lower latency and better scalability than polling operation through HTTP. A WebSocket service is easily implemented in Java through annotations described in JSR356 - "Java API for WebSockets" [13]. The server class does not inherit from `HttpServlet` and the client certificate is therefore not readily accessible since the `HttpServletRequest` object is not within scope.

The WebSocket service is (possibly through annotation processing) still running as a Servlet and some servlet containers (i.e., Tomcat and Glassfish) will allow Servlet Filters to be inserted in front of it, giving access to the `HttpServletRequest` object from there. Furthermore, the `HttpSession` object can be used to convey information into a `ServerEndpointConfig.Configurator` object, which can move the desired information into the `UserProperties` object (found through the `ServerEndpointConfig` object). The WebSocket server class can pick up this information in the `@OnOpen` method through the `EndpointConfig` object.

Quite a few problems arose during the WebSocket based experiment. Setting up the TLS configuration on Glassfish revealed that the configuration console is not working properly, and manual editing of configuration files (a poorly documented process) was necessary. The Glassfish server has a large installation footprint but worked otherwise as expected with regards to the arrangement presented in the previous paragraph.

The Jetty server (version 9.3.8) supports WebSocket through the JSR356 API and is easily configured for client authenticated TLS. It is, however, not possible to insert a Servlet Filter in front of the WebSocket server object, and no other way to access the client certificate was found. Additionally, Jetty refuses to set up TLS protected WebSocket connections (through the `wss://` protocol prefix) from a Javascript client running in Chrome (contrary to Firefox). Jetty was therefore deemed unsuited for our demonstrator application.

The Tomcat server (version 8) also implements WebSocket JSR356 API, and supported the presented arrangement for retrieval of the client certificate. The installation footprint is smaller than Glassfish (since this is not a full J2EE server) and a standalone configuration with automatic WAR-file deployment was quite easy to set up. Tomcat is therefore regarded as the best alternative for applications that seek to employ the presented security arrangement for WebSocket communication.

For the selection of a client certificate during the establishment of a TLS connection initiated by the `XmlHttpRequest` object or an ordinary page loading operation, all the browsers used in this experiment have prompted the user with a list of available certificates to choose from. For subsequent connections, this certificate will be used for the same server, also for connection made through the `WebSocket` object (using the `wss://` protocol prefix). On the other hand, the `WebSocket` object does not itself prompt the certificate selection dialogue, and the connection will fail if there is no existing association between certificate and server in the client browser (created by a page load or the `XmlHttpRequest` object). During the application design, one must assure that a client authenticated TLS connection is established (by the `XmlHttpRequest` object or an ordinary page loading operation) before the first WebSocket TLS connection so that the necessary association is created.

D. Interoperability and performance observations

The presented client code was tested on several platforms and on several browsers and their interoperability properties were observed. The client candidates were:

- Google Chrome on ChromeOS, Linux, Android and MacOS
- Mozilla Firefox on MacOS and Linux
- Apple Safari on MacOS

The results were encouraging: Firefox required an extra CORS header (`Access-Control-Allow-Credentials`) in the HTTP response to allow client-authenticated TLS connections. Otherwise, Chrome and Firefox both executed the application without differences. Safari was not able to run the application for several reasons, the main one being an immature implementation of WCA lacking support for the chosen cryptographic algorithms.

V. Conclusion and Remaining Research

This paper has investigated the necessary mechanisms for the provision of attested integrity for cloud security. The attests provide trust in the correctness of the client platform and client application code. It has been shown that remote attestation and device-bound private keys are necessary elements, and that Google's ChromeOS provides the most complete solution for platform trust. Together with the CORS protocol and TLS communication protection, it is feasible (with a few identified reservations) to obtain trust in the client-side application software as well.

The demonstration application offers a chat forum of signed messages based on these mechanisms. The exchanged messages use the JSON syntax with digital signatures as defined by RFC 7515 [14]. The demonstrator has identified two major shortcomings in the WCA definition and library availability:

1) There is no way to import keys from the browser's key store into WCA, and the application had to import keys from less protected channels (HTTPS connections or the local file system)
2) There is no support for SOAP-based security objects, like XML-DSIG or XML-ENC, and derived objects like WSS, SAML, etc. on the client side.

Finally, the ability to trust the application code for correctness alleviates the well known *what you see is what you sign*

```
public class CrossSiteGuard implements Filter {
    PublicKey caPubKey = ... // loaded from internal resource
    final static String csName = "https://cs.ffi.no:8443";
    final static String chromeOSindicator = "OU=ChromeOS";
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        if (request instanceof HttpServletRequest) {
            HttpServletRequest req = (HttpServletRequest)request;
            HttpServletResponse resp = (HttpServletResponse)response;
            String method = req.getMethod();
            X509Certificate[] clientCert =
            (X509Certificate[])req.getAttribute("javax.servlet.request.X509Certificate");
            if (clientCert == null) resp.sendError(401,"Client␣authentication␣is␣required");
            String clientDN = clientCert[0].getSubjectX500Principal().getName();
            try { clientCert[0].verify(caPubKey); // Throws exception if not ok
            } catch (Exception ce) { throw new ServletException("Illegal␣certificate␣issuer"); }
            if (clientDN.contains(chromeOSindicator)) request.setAttribute("no.ffi.anf.trust", "high");
            else request.setAttribute("no.ffi.anf.trust", "low");
            if (method.equals("OPTIONS")) {
                resp.addHeader("Access-Control-Allow-Headers", "Content-type");
                resp.addHeader("Access-Control-Allow-Origin", csName);
                resp.addHeader("Access-Control-Allow-Credentials", "true"); // For Firefox
            } else if (method.equals("POST")) {
                resp.addHeader("Access-Control-Allow-Origin", csName);
                resp.addHeader("Access-Control-Allow-Credentials", "true");
            } else if (method.equals("GET")) {
                resp.addHeader("Access-Control-Allow-Origin", csName);
                resp.addHeader("Access-Control-Allow-Credentials", "true");
            }
        }
        chain.doFilter(request, response);
    }
...
```

Fig. 3. Java source code for integrity attestation control through a Servlet filter

(WYSIWYS) problem, well explained in [15]. The WYSIWYS problem concerns the user's ability to ensure that the signed object really contains the data that the user intends to sign, and that the private key is not leaked during the process. Although unintended modification of the object, as well as a leaked key, can happen due to vulnerabilities in the original software, the presence of malware that affects the signature operation is a more plausible cause. It is likely that the WYSIWYS problem becomes less acute if one can ensure that the client software (which generates the signature) is integrity protected, inspected and approved by the owners of the related service. The same advantage can apply to clients who read sensor data, as long as the connection between the sensor and the computer is protected: The sensor readings can be trusted not to have been modified by malware or rogue client software.

REFERENCES

[1] N. A. Schear *et al.*, "Secure and resilient cloud computing for the department of defense," *Lincoln Laboratory Journal*, vol. 22, no. 1, pp. 123–135, 2016, https://www.ll.mit.edu/publications/journal/pdf/vol22_no1/22_1_10_Schear.pdf [Online: accessed Oct 2020].

[2] U.S. Department of Defense, "DoD Moves Data to the Cloud to Lower Costs, Improve Security," https://www.defense.gov/News/Article/Article/604023, [Online: accessed Oct 2020].

[3] World Wide Web Consortium (W3C), "Web cryptography api," http://www.w3.org/TR/WebCryptoAPI/, [Online: accessed Oct 2020].

[4] A. Fongen and F. Mancini, "The integration of trusted platform modules into a tactical identity management system," in *IEEE MILCOM*, San Diego, USA, 2013, pp. 1808–1813.

[5] Trusted Computing Group, "TPM Main Specification," http://www.trustedcomputinggroup.org/resources/tpm_main_specification, [Online: accessed Oct 2020].

[6] Google, "Verified Boot," http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot, [Online: accessed Oct 2020].

[7] ARM, "ARM Security Technology - Building a Secure System using TrustZone® Technology," 2009, white Paper.

[8] World Wide Web Consortium (W3C), "Same origin policy," https://www.w3.org/Security/wiki/Same_Origin_Policy, [Online: accessed Oct 2020].

[9] I. Yusof and A. S. K. Pathan, "Mitigating cross-site scripting attacks with a content security policy," *IEEE Computer*, vol. 49, pp. 56–63, 2016.

[10] World Wide Web Consortium (W3C), "Cross-Origin Resource Sharing," https://www.w3.org/wiki/cors/, [Online; accessed Oct 2020].

[11] A. Cunningham, "Chrome os management console brings improvements for businesses," http://arstechnica.com/information-technology/2012/06/chrome-os-management-console-brings-improvements-for-businesses/, [Online: accessed Oct 2020].

[12] Y. Sheffer, R. Holz, and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)," IETF RFC 7457, Oct. 2015.

[13] Oracle corp., "JSR 356, Java API for WebSocket," http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html, [Online: accessed Oct 2020].

[14] N. Sakimura, M. Jones, and J. Bradley, "JSON Web Signature (JWS)," IETF RFC 7515, Dec. 2015.

[15] P. Landrock and T. P. Pedersen, "Wysiwys? - what you see is what you sign?" *Inf. Sec. Techn. Report*, vol. 3, no. 2, pp. 55–61, 1998, http://dx.doi.org/10.1016/S0167-4048(98)80005-8 [Online: accessed Oct 2020].

 ISBN: 978-1-61208-821-1