SGX Project

Team Awesome

August 10, 2016

Abstract

Web Servers are now often deployed in the cloud environment. To secure communications in the middle of the network between clients and web servers, the Internet community has come to rely on the SSL/TLS protocol which requires a strong secret (private key) to be stored with the server. However, the server administrator, in this model, is forced to trust the cloud provider to not disclose their private key. We propose a modification to legacy web servers and cryptographic libraries which secures the sensitive key material in the face a malicious provider or a compromised operating system without the need to trust the cloud provider and/or operating system.

Contents

1 Introduction **4**

2 Background **6**
2.1 An overview of Intel SGX 6
2.2 Threat model 6
2.3 Provisioning the enclave with the long-term private key 7
2.4 Inter-platform attestation and secret provisioning 8
2.5 SSL/TLS Overview 9

3 Design **10**

4 Implementation **12**

5 Project Management **12**

6 Conclusion **13**

1 Introduction

With the recent surge in privacy concerns employing SSL/TLS to secure communications in the middle of the network has become common place. SSL/TLS offers guarantees of confidentiality and integrity provided that a private key's secrecy is maintained. Yet, SSL/TLS was designed assuming that its user trusts the hardware and OS of the machine on which the key is held.

While this assumption is perfectly valid in the case where a person is running an SSL/TLS enabled service on their own machines, many web applications are now hosted by third party cloud service providers such as Amazon Web Services, Heroku, Digital Ocean &c. Moreover, to offer SSL/TLS, the private key must also be stored with the web application on these service providers' machines. This implies that a server administrator using the aforementioned services is trusting the cloud-provider, including any personnel with physical/administrative access to the machines, and the underlying OS to maintain the secrecy of the sensitive key material. Such a wide trust surface makes it difficult to maintain the privacy of critical secrets.

Consider a case where the cloud provider is not malicious; a vulnerability within their platform could lead to leaking the private key, if exploited by an adversary. Moreover, if the cloud provider is indeed malicious they could simply read your private key from the hard disk, if stored unencrypted, or mount some form of memory sniffing attack to read the key from the web server's memory since data in RAM is unencrypted. A compromised private key allows an adversary to do the following:

- Decrypt past, stored, communication between the web server and a client (assuming a cipher that does not provide perfect forward secrecy is in use)
- Decrypt any ongoing communication between the web server and a client
- Masquerade as the server and fool a client into disclosing sensitive information such as passwords

In all cases, a compromised key voids the confidentially and integrity guarantees of SSL/TLS.

Our project aims to break this assumption by securing the private key in the face of: An adversary who is capable of exploiting the server application, a malicious cloud provider, and an adversary with an exploit for the underlying operating system.

The remainder of this report is divided as follows: Section 2 provides an overview of previous work and technologies underlying our project, Section 3 discusses the design of the system that we implemented to meet the above-stated goals, Section 4 highlights some of the implementation considerations that we had to make in realizing the system that we designed, Section 5 details the managerial aspects of this project, and, finally, Section 6 offers areas where this project could be improved, and concludes this report.

The first of these goals has been extensively addressed in previous works including Wedge [1] and OKWS [2]; the latter two have hardly been addressed before, and are key points in recent works such as Haven [3].

2 Background

2.1 An overview of Intel SGX

This section will not cover all of the details of SGX but only those applicable to our project, for a complete treatment of SGX please refer to [4]. Intel SGX is a set of x86 instructions that allow for a programming model wherein a program can be split into two components: an untrusted component that executes as normal and a trusted component that executes within a protected area of RAM, called an enclave, which can only be accessed when executing the trusted component.

The protection of an enclave is managed by the CPU; any data written to the enclave is encrypted first by a memory encryption engine (implemented in hardware) and is only decrypted when required by the CPU during the execution of the trusted component for which that enclave belongs. The key used for this encryption process is derived from a combination of a device key, unique to each SGX-enabled CPU and the "identity" of the enclave called MRENCLAVE, a cryptographic hash of the enclave's contents at the trusted component's initialization. SGX thus ensures that no process other than the one that initialized the enclave can access the protected area.

Interacting with the trusted component, as a result, may only occur through invoking a programmer defined interface, called a callgate, as depicted in Figure 1.

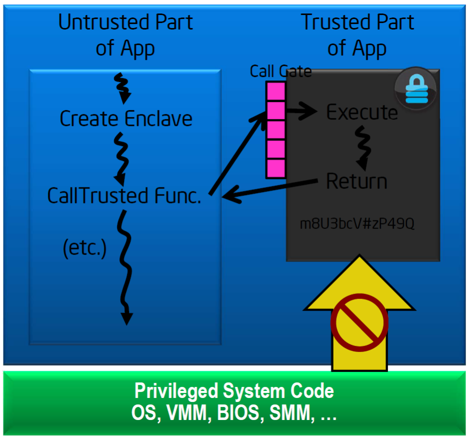

Figure 1: Interaction of the untrusted part of the application with the trusted part can only occur through a callgate

2.2 Threat model

We consider a threat model similar to the one presented in Haven [3]. Our TCB includes a correctly implemented and SGX enabled CPU and all instructions executing and data resident within an enclave. Therefore, we assume an attacker cannot access the SGX processor key provisioned by Intel itself, which is used to generate subsequent cryptographic keys that preserve the confidentiality, integrity and authenticity of an enclave.

An adversary can take full control of everything beyond the TCB. That is, we assume all software executing on the platform (outside of the enclave), the underlying operating system, the hypervisor, all firmware and the BIOS are potentially compromized. Side-channel attacks that originate from other sources such as CPU cache timing information (L1, L2, L3), power consumption or other entropy source are considered as out of scope of this work. Finally, we assume an attacker may act as man-in-the-middle to eavesdrop active sessions and, as further analyzed in the Limitations section, launch a denial-of-service attack, though without compromizing any secret isolation guarantees of our design.

2.3 Provisioning the enclave with the long-term private key

Provisioning a web server with a long-term private key for the purposes of SSL/TLS is currently done storing the private-key along with the executable on the remote machine. This scheme, however, assumes a trusted cloud-provider/OS. However, under our threat model this scheme is not viable. Alternatively, we require a method by which we can verify the identity and integrity of the server application and then, upon successful verification, we can send the long term private key to the server in a secure fashion. Loosely, the requirements are as follows:

- The mechanism allows the verification of the identity and integrity of the server application and the underlying TCB. This is so that we can be sure the private-key is being sent to the same server we placed on the remote machine, and the software is being executed by trustworthy hardware.
- The mechanism allows us to setup a secure channel, ensuring that the only entities privy to the private key are the server application and the server administrator.
- The mechanism allows for the verification of the entity providing the private key. If this requirement were not in place, the mechanism could allow any arbitrary entity to authenticate with the server and provide their own private key. Observe that such an attack does not compromise the security of the long-term secret, but makes it possible to render the server useless (if the matching public certificate is not placed onto the server, verifying the server's hostname, then clients will reject connections to the server under the SSL/TLS protocol).

The first and second requirement are met by a process called inter-platform attestation, outlined by Intel in [4] and summarized in the following section.

2.4 Inter-platform attestation and secret provisioning

Inter-platform attestation is a mechanism that can be invoked by an entity, referred to as the challenger, running on one platform to verify an enclave running on another, remote, platform. This process enables the challenger to verify the following about the remote enclave:

1. The contents of the enclave's pages (code, data, stack and heap) upon creation (after the ECREATE instruction completes)
2. The identity of the entity that signed the enclave
3. The trustworthiness of the underlying hardware
4. Authenticity and integrity of any data generated by the enclave and sent as part of the attestation process. This allows us to satisfy the second requirement by generating an ephemeral key pair and binding it to the remote attestation process. This, therefore, allows the challenger to verify the integrity of the ephemeral public key and verify that it was generated by the server application.

The steps involved in the attestation process are as follows (illustrated in Figure 2):

1. The challenger invokes the remote attestation mechanism to verify the identity and integrity of the remote enclave
2. The non-trusted part of the web server receives the challenge, passes it along to the trusted portion of the web server along with the identity of the quoting enclave. The quoting enclave is a special enclave provided by Intel as part of the SGX platform to enable remote attestation by verifying the integrity of the underlying hardware.
3. The enclave invokes EREPORT which is an SGX instruction that generates a REPORT structure to be provided to a *local* enclave, the quoting enclave in this case. This structure contains a hash of the contents of the enclave's pages upon ECREATE's termination, a hash of the identity of the enclave's signer, a hash of any user-data, the ephemeral key in our case, generated by the enclave. The REPORT is signed by a MAC-key that can only be accessed by the CPU and the quoting enclave. The REPORT along with the ephemeral key is then sent to the non-trusted part of the application.
4. The REPORT is sent to the quoting enclave where its integrity is verified by calculating the MAC across its contents.
5. Assuming the REPORT is verified successfully, the quoting enclave generates a QUOTE structure that includes the REPORT structure and a signature across the quote generated using a key known as the EPID key. The EPID key is a private key unique to the CPU that is part of the platform and verifies the firmware of the processor and its SGX capabilities.

6. The QUOTE is sent along with the ephemeral key to the challenger

7. The challenger verifies the QUOTE structure by using an EPID public certificate. If this is successful then the challenger is sure that this QUOTE came from a valid SGX CPU and can trust its authenticity. The challenger can then check the contents of the REPORT contained within the QUOTE to verify the identity of the remote enclave, and the integrity of the ephemeral key received along with the QUOTE. The ephemeral key, if proven to be valid, can now be used to communicate with the remote enclave in a secure manner.

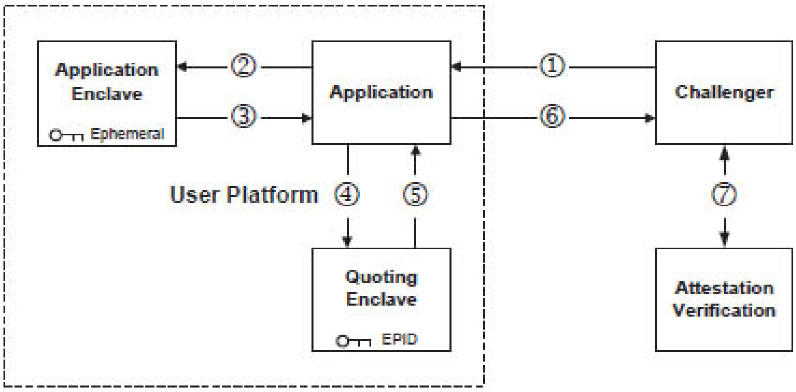

Figure 2: Remote Attestation and Secret Provisioning

However, the process outlined above takes no steps to verify the challenging entity to the trusted component. In an effort to authenticate the challenger, we utilize a combination of asymmetric cryptography and SGX's guarantees. First, the server-administrator, before shipping the server program to the cloud-provider, stores the public key for the challenger within the text-area of the trusted component. The reason for this comes from realizing that doing so binds the public key to the identity of the trusted component, and as a result, the SGX-enabled CPU would not start the trusted-component if the public key has been tampered with. All that remains now is for challenger to sign the long term private key with their own key, enabling the trusted-component to verify it using the challenger's public-key that was shipped along with the server-program.

2.5 SSL/TLS Overview

3 Design

By utilizing the aforementioned remote-attestation process, we can provision the trusted component of the remote server application with an SSL private key while making it extremely difficult* for even *privileged* processes running on the cloud provider to access the key. This is a stark contrast to current schemes wherein the private key is merely shipped as part of the server application to the cloud provider.

As previously highlighted in Section 2.5, the private key is required by a subset of the operations executed during the handshake step. These operations have to be executed within the trusted component, and are invoked by the non-trusted component via an interface that we define in this section. Note that the interface has to be carefully designed, allowing the handshake to complete correctly while not exposing the long term private key through an oracle.

Take for example an interface where the non-trusted component supplies (`ServerRandom`, `ClientRandom`, `{PremasterSecret}`$_K$). Such an interface, while maintaining the secrecy of the private key's bits, would allow an adversary capable of exploiting the non-trusted component to generate the symmetric keys for previously eavesdropped sessions. Therefore, it is no better than leaving the private key in the non-trusted component. However, observe that it is not necessary for `ServerRandom` to be provided by the non-trusted component, it need only be provided by the *server*.

We can adjust the interface so that the non-trusted component supplies (`ClientRandom`, `{PremasterSecret}`$_K$), both of which are generated by the client, and the trusted component generates a new `ServerRandom` every time the interface is invoked. The resulting interface ensures that, even if a previously eavesdropped `{PremasterSecret}`$_K$ is provided, a fresh session-key is computed on every invocation.

We consider two scenarios:

- Session keys available to the OS
- Session keys hidden inside the enclave and accessible through encrypt/decrypt oracle

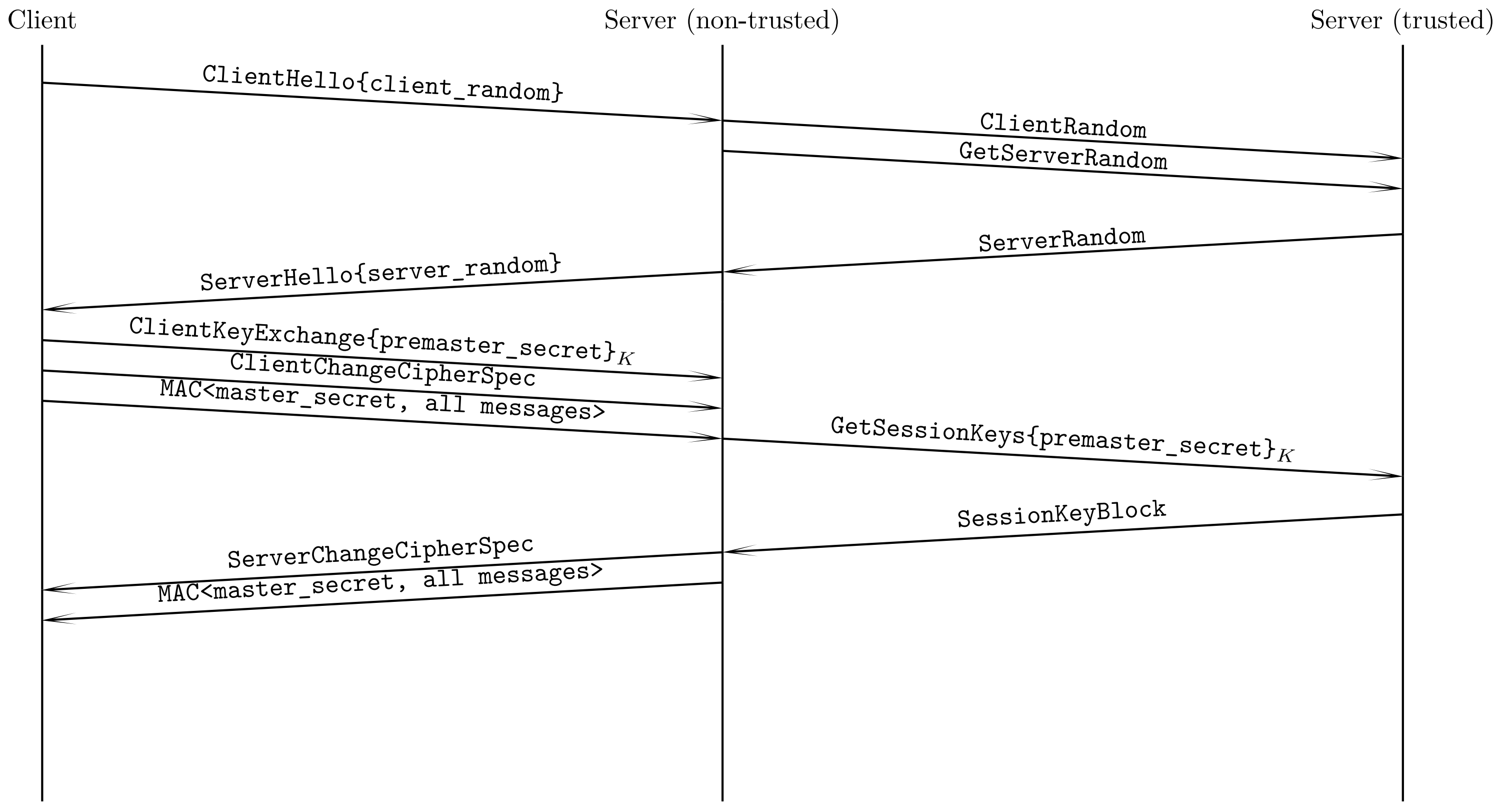

Figure 3: SSL handshake

ECDHE Handshake TBC..

4 Implementation

5 Project Management

Hello world!

Hello, here is some text without a meaning. This...

6 Conclusion

Hello world!

Hello, here is some text without a meaning. This...

References

[1] Andrea Bittau et al. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *NSDI*. 2008.

[2] Maxwell Krohn. "Building secure high-performance web services with OKWS". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. 2004. URL: `http://portal.acm.org/citation.cfm?id=1247415.1247430`.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283. ISBN: 9781931971164. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann`.

[4] Intel Corporation. *Intelő Software Guard Extensions Evaluation SDK for Windows* OS*. 2010. URL: `https://software.intel.com/sites/default/files/managed/d5/e7/Intel-SGX-SDK-Users-Guide-for-Windows-OS.pdf`.