A Practical TLS Enabled Web Server Without Trusting The Operating System

A. Awad C. Chaiphet I. Gakos W. Grajkowski
B. Jacob

August 27, 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 of a malicious provider, or a compromised operating system, without the need to trust the cloud provider and/or operating system.

Contents

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, assuming the key is not encrypted, or mount some form of memory sniffing attack to read the key from the web server's memory since data in RAM is not encrypted. 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.

Clearly, implicitly trusting a cloud-proivder and the OS with sensitive key material poses a considerable security risk. Yet, this assumption of trust is usually the case when server administrators deploy web applications via cloud based services. Our project aims to break this assumption by refactoring legacy web servers to secure the long term private key in the face of: (1) An adversary who is capable of exploiting the server application, (2) a malicious cloud provider, and (3) an adversary with an exploit for the underlying operating system. We utilise the principle of least privilege and along with a recently released technology from Intel® called Software Guard Extensions, both of which are described in Section 2, in designing our system.

The remainder of this report is divided as follows: Section 2 provides a detailed overview of previous solutions to the problem outlined here, 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 a few implementation considerations

in realising the system that we designed, Section 5 presents the evaluation we have conducted of the system, Section 6 details the managerial aspects of this project, and, finally, Section 7 offers areas where this project could be improved, and concludes this report.

2 Background

2.1 An Overview of SSL/TLS

Before discussing previously proposed solutions to the problem we identified in the Introdcution, we present here a generic overview of SSL/TLS.

SSL/TLS supports a wide array of ciphers, but they can be roughly split into two categories: ciphers that support forward secrecy, and ciphers that do not. Ciphers that offer forward secrecy are ones wherein a compromised long-term private key does not allow an adversary to compromise previously eavesdropped, stored sessions. SSL/TLS's session establishment mechanism is different based on the category to which the cipher belongs. Consequentially, depending on the type of cipher used for a given session, our system must adapt to two, different, control flows. In this section we present an abstraction of the SSL/TLS handshake that we heavily referred to in designing our system.

In general, regardless of the cipher used, SSL/TLS utilises the long-term private key to negotiate ephemeral keys for use in the current session; hence, to secure the long-term private key, we need to examine the session establishment mechanism. Roughly, SSL/TLS's handshake can be split into four steps:

1. **Contacting the server and establishing parameters for the session**. Specifically, the exchange of hello messages that include: the server's certificate(s) a list of ciphers supported by each side (to determine which cipher is to be used for this session), `ServerRandom`, and `ClientRandom`. The random variables are used to derive the secret, used to secure communications.

2. **Asymmetric key exchange**: based on the cipher selected, the client and server determine the asymmetric keys that are to be used to exchange the symmetric keys for the session. This step is necessary in two scenarios:

   (a) The client and server both possess public key certificates, and both entities wish to verify each other's identity during the handshake. We do not consider this case due to its rarity. Generally, the client verifies the server's identity as part of the SSL/TLS handshake, and the server verifies the identity of the client via some other means, such as a username & password.

   (b) The client and server selected a cipher that offers forward secrecy. These ciphers function by first exchanging an ephemeral asymmetric secret. This secret is then used in negotiating the symmetric secret.

   In all other cases, the server's asymmetric keys are used to negotiate the ephemeral symmetric secret.

3. **Ephemeral symmetric key negotiation**: The client and server establish the symmetric secret that is used for the current session. The symmetric ephemeral keys are calculated as outlined below:

   (a) The `ClientRandom`, `ServerRandom`, and a value denoted `PremasterSecret` are combined together, through use of a pseudo-random function (PRF), to generate a value called the `MasterSecret`. The `MasterSecret` is a 48-byte

number, and is computed using the method outlined here in all SSL/TLS ciphers. In contrast, the `PremasterSecret` is a random value, established as part of this step, the derivation of which depends on the cipher used.

(b) The `MasterSecret` is then used to generate a key block. A session key block consists of:

- `server_write_key`: this key is used by the server to encrypt outgoing packets, and by the client to decrypt incoming packets.
- `client_write_key`: this key is used by the client to encrypt outgoing packets, and by the server to decrypt incoming packets.
- `server_mac_secret`: this key is used by the server to compute a MAC over outgoing packets, and by the client to verify the MAC over incoming packets.
- `client_mac_secret`: this key is used by the client to compute a MAC over outgoing packets, and by the server to verify the MAC over incoming packets.
- `client_initialisation_vector(iv)`: this is not a key, but a value used to initialise the symmetric cipher at the client, before invoking the encryption routine on outgoing packets, and at the server, before invoking the decryption routine on incoming packets.
- `server_iv`: same as above, but used by the server before invoking the encryption routine on outgoing packets, and by the client before invoking the decryption routine on incoming packets.

4. **Verifying the integrity of the just-negotiated keys, completing the session establishment**: Both the server and the client compute a MAC across the packets exchanged in establishing the session. The resultant finished message is encrypted using the just-negotiated keys. If both sides successfully verify the MAC, the handshake is complete and the session is established. If either side fails to verify the MAC, the session is terminated.

Figure 1 illustrates the generalization of the SSL/TLS handshake we presented here.

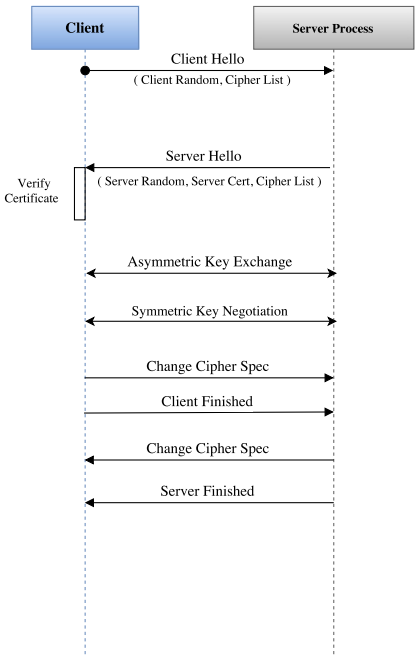

Figure 1: SSL/TLS handshake generalization

2.2 Previously Proposed Solutions

2.2.1 The Principle of Least Privilege

The first of our goals has been extensively addressed in previous works [1, 2] through use of the principle of least privilege. The principle of least privilege (PoLP) requires that components of a system operate with the minimum resources required to complete their respective tasks. PoLP is heavily utilized in systems security research to design systems that maintain their integrity in the face of an adversary capable of exploiting some of their components. In contrast, the most popular web servers today execute as monolithic applications, including Apache and NGINX, where all of their processes have the same level of privilege and access to the sensitive key material. Exploiting any one of these processes may therefore lead to leaking the private key.

To motivate the design of our system, we begin by discussing a system called Wedge [1] that was used to refactor Apache, by enforcing PoLP, securing the private key in the face of an adversary who is capable of exploiting the web server application. Wedge's authors concentrated their efforts on SSL/TLS ciphers that do not offer forward secrecy, and the scenario where only the client verifies the server's identity as part of the SSL/TLS handshake. Figure 2 illustrates the resulting SSL/TLS handshake. We refer to this handshake as the "RSA" handshake for brevity[1]. The asymmetric key exchange step is not necessary here, as mentioned in the previous section. The symmetric key negotiation step comprises of the client generating a random value for `PremasterSecret`, encrypting it in

[1] The cipher usually used in this handshake is RSA; hence the name

the server's public key, and sending the resulting packet to the server. Observe that the long-term private key, as a result, is only used in decrypting $\{\texttt{PremasterSecret}\}_K$.

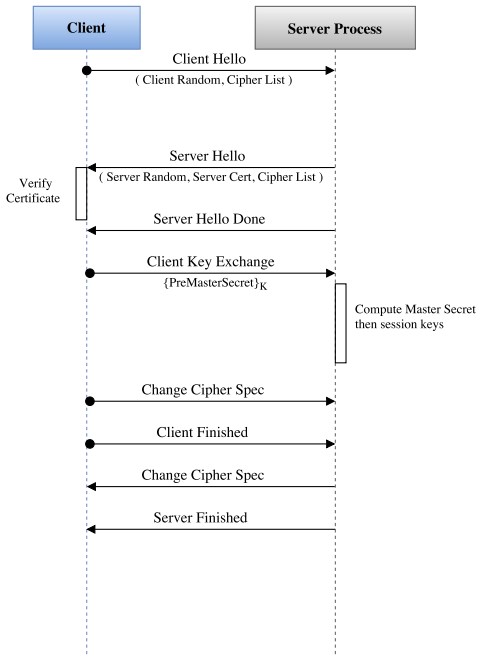

Figure 2: SSL/TLS handshake for ciphers that do not offer forward secrecy. *K* here is the server's public key, and the braces around `PremasterSecret` indicate the packet is encrypted

Moving forward, we list the attacks possible against a monolithic web server, and then detail how the design implemented with Wedge resolved these vulnerabilities. In this section, we only consider a threat model where the adversary is capable of passively eavesdropping on secure communication channels, and exploiting unprivileged components of the server. We did not consider the second threat model detailed in the Wedge paper, wherein an attacker is capable of actively modifying packets exchanged between the web server and client, because the solution presented in there does not hold if we remove the assumption that the OS and cloud provider are trusted. This is a consequence of the web server's user-level processes (dynamic content generation scripts, databases &c.) requiring data in plain-text to complete their tasks. As such, even if we secure the encrypt/decrypt interface using the solution in the Wedge paper, plain-text data would be available in a non-privileged process's memory. This is further discussed in Section 3.

Possible Attacks

There are two main attacks that may be mounted by an adversary capable of exploiting *only* unprivileged components of the web server application and passively eavesdropping on packets exchanged between the server and the client:

1. The adversary could leak the private key from the network facing worker process. This could be done by reading the private key from disk directly or from the process's memory space. The attack is illustrated in Figure 3.

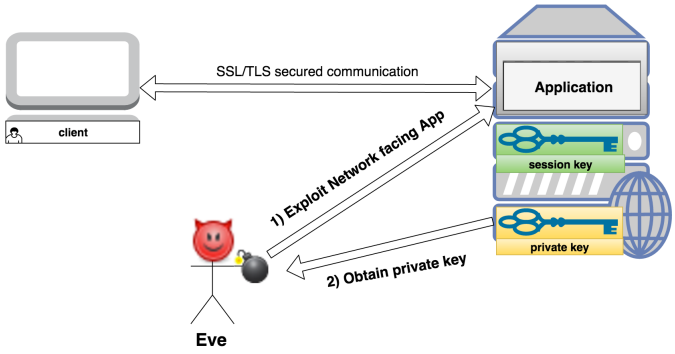

Figure 3: Exploiting the network facing component to leak the private key

2. The adversary may record traffic exchanged over the SSL/TLS channel and then exploit a naive session key generation interface to acquire the session keys used in that exchange. A naive interface is one that accepts `ClientRandom`, `ServerRandom`, and $\{\texttt{PremasterSecret}\}_K$ where K is the web server's public key. Such an interface allows an adversary, after exploiting the unprivileged component, to generate any previously eavesdropped session's symmetric key and is, therefore, no different to having read access to the private key from the adversary's perspective. This exploit only works if the cipher used does not offer forward secrecy, and it only affects the single session which was eavesdropped. The attack is illustrated in Figure 4.

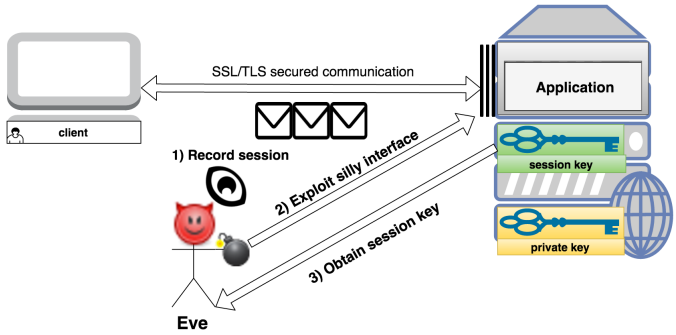

Figure 4: Exploiting the network facing component and the naive session key generation interface to generate session keys for eavesdropped session

Proposed Solution

The solution proposed in the Wedge paper is achieved through partitioning the session key generation code into its own logical compartment [2] that executes at high privilege. This partitioning can be seen in Figure 5.

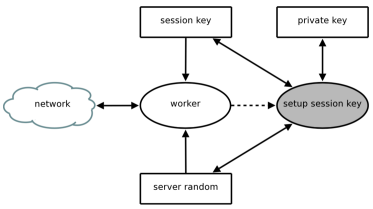

Figure 5: Partitioning Scheme to protect against leaking the private key [1]

The low privilege worker process does not possess the private key, but must rely on an interface to the session key generation process. The interface takes `ClientRandom` and $\{\texttt{PremasterSecret}\}_K$ as arguments, both of which are provided by the client, while the high privilege component generates a fresh `ServerRandom` upon invoking the interface. The freshness property ensures that, even if an adversary exploits the worker process, previous session keys cannot be generated by simply invoking the session key generation interface.

While this design secures the private key against the previously described adversary, it is not, by itself, sufficient to secure the key against an adversary capable of exploiting the OS, or from a malicious cloud-provider with full access to the physical machine. Such adversaries may leak the key from the privileged component's memory space by exploiting the OS, or, in the case of the malicious cloud provider, simply read the key from disk[3]. A hardware component, called a trusted platform module, has been utilised in an effort to secure the private key against the adversaries described above.

2.2.2 Trusted Platform Modules

Securing the private key against an adversary capable of exploiting the operating system is achievable through use of specialised hardware called a trusted platform module (TPM). TPMs can secure the private key through encryption via a Storage Root Key (SRK) [3]. The SRK's integrity is maintained by ensuring that its private component may never leave the TPM. As a result, the long term private key itself can never be decrypted outside the TPM. The TPM is also capable of executing cryptographic operations, including those that make use of the long term private key in SSL/TLS. Consequentially, by delegating all private key operations to the TPM, one can rest assured that their private key cannot be compromised without compromising the TPM itself.

[2] This compartment is called an sthread in Wedge's nomenclature

[3] Even if the long-term private key is encrypted on disk, the key for decrypting it must be present in the binary file of the privileged component (which is unencrypted), or in the process's memory at runtime in plain-text

The security benefits of a TPM, however, were outweighed by the cost of purchasing the additional piece of hardware, and TPMs did not gain any traction with cloud providers. In this project we utilized a new technology from Intel® called Software Guard Extensions (SGX). SGX is an augmentation to Intel®'s ISA which offer the ability to launch encrypted regions of memory, called enclaves, where only trusted regions of code can read/write. This allows for TPM-like functionality, but SGX has the advantage of being deployed as part of new, commodity, CPUs released by Intel®. As a result, when cloud providers upgrade their machines, they will possess the ability to support SGX programs without purchasing additional hardware.

SGX has gained momentum as a research platform for security related work such as Haven [4], which secures a legacy application from a non-trusted OS and cloud-provider *without* modifying the application's source code. Yet, there is no work, to our knowledge, that attempts to secure only the private key material through use of SGX. Narrowing the trusted region to contain only the component that handles the private key allows us to define a much smaller trusted computing base that only contains the CPU and the code that handles the long term private key.

2.3 An Overview of 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 [5]. 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.

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, which we refer to as the *enclave program*, to 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 (MRENCLAVE), a cryptographic hash of the enclave's contents at the *enclave program*'s initialization. SGX thus ensures that no process, other than the one that initialized the enclave, may access the protected area.

Interacting with the *enclave program*, as a result, may only occur through invoking a programmer defined interface, called a callgate, as depicted in Figure 6.

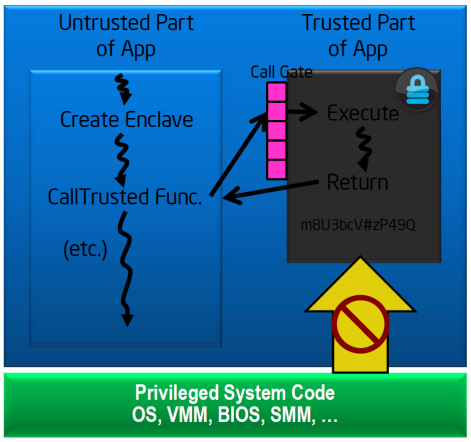

Figure 6: Interaction of the untrusted part of the application with the trusted part can only occur through a callgate

2.3.1 Initialising an SGX Enclave

Launching an SGX *enclave program* requires the execution of the following steps:

1. The *enclave program* is first compiled and then signed with an Intel® verified certificate[4]. The signing process generates a data structure called `SIGSTRUCT` that contains the aforementioned MRENCLAVE, the identity of the signing entity, MRSIGNER, and the signature across this structure.

2. When attempting to launch the *enclave program*, the CPU calculates MRENCLAVE using priveleged SGX instructions, calculates MRSIGNER by taking the hash of the public key contained in the signing entity's certificate, and then attempts to verify that MRSIGNER_{calc} is equal to $\text{MRSIGNER}_{SIGSTRUCT}$, MRENCLAVE_{calc} is equal to $\text{MRENCLAVE}_{SIGSTRUCT}$, and that the signature across `SIGSTRUCT` was generated by the signing entity indicated by MRSIGNER.

3. If all previous checks complete successfully, then the CPU proceeds to execute the *enclave program* otherwise, the CPU aborts execution.

At the time we begun working on this project we did not posses access to SGX hardware, forcing us to use a simulator. There were two choices, at the time: OpenSGX and the Intel® Windows SDK's simulation mode. We selected the former to implement our prototype as it was compatible with Linux, a platform we were more familiar with, and had several examples we could refer to for guidance in our implementation (the Intel SDK, at the time, was under-documented). The following section provides a quick examination of OpenSGX's salient features.

[4] While the requirements imposed on the certificate for successful verification are well documented, the process is not automated, and involves an application to Intel®'s Infrastruction Attestation Service (IAS)

2.3.2 Brief Notes on OpenSGX

OpenSGX [6] is a platform that provides a software emulation layer enabling the execution of SGX instructions *without* the possession of SGX hardware. Specifically, OpenSGX implements a hardware emulation layer, implementing SGX's instructions and data structure as per Intel®'s specification [5], and an OS emulation layer, providing access to wrapper functions for privileged SGX instructions (instructions, such as EADD and EINIT, used to bootstrap an enclave fall under this category)[6].

Programs that run on top of OpenSGX consist of the trusted code and a wrapper program which initializes the enclave and provides the trusted code with an interface to `sgx-lib`, a library that contains wrapper functions for user-level SGX instructions.

Intel®'s SGX specification does not detail how an enclave program may invoke system calls[7]; OpenSGX specifies an interface which allows an enclave program to invoke `libc` functions. The interface works as follows:

1. The enclave program writes the arguments for the system call into a shared area of memory called a *stub*
2. The enclave program invokes EEXIT, exiting enclave mode, and executes a predefined handler called a *trampoline* which invokes the system call
3. After the system call is complete, results are written to into the *stub* and ERESUME is called, resuming the execution of the enclave program

By creating this interface, writing code for OpenSGX is akin to writing C code, making the development process familiar. Figure 7 illustrates the high level design of OpenSGX.

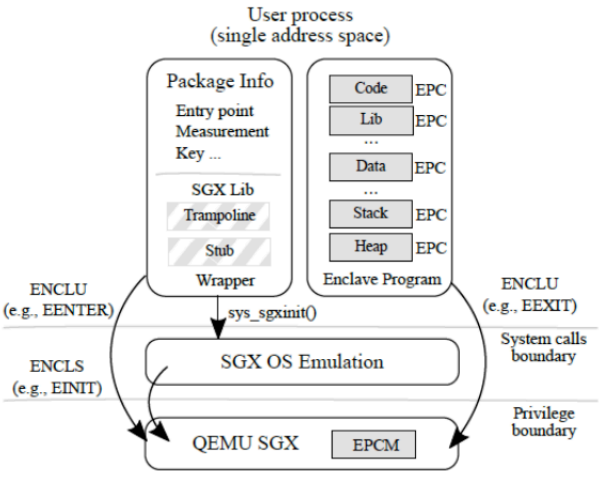

Figure 7: High level design of OpenSGX. Grey boxes indicate protected areas of memory. Striped boxes indicate shared areas.

OpenSGX also provides a tool-chain for automating tasks such as compiling and signing binaries, generating signing keys, and verifying and executing correctly signed

binaries. However, it is imperative to note that OpenSGX does not provide the same security guarantees as SGX; it does not encrypt the enclave memory region, for example. Yet, it has proven to be useful as a platform for testing the design of our system, and measuring its performance.

2.4 Provisioning The Enclave With The Long Term Private Key

SGX by itself, however, does not answer an imperative question: How do you get the private key into the enclave in the first place? While the key would be encrypted as it is used by an *enclave program* in RAM, it cannot be simply shipped as part of the binary. This is because the binary is not encrypted. Yet, provisioning a web server with a long-term private key for the purposes of SSL/TLS is currently done by storing the private-key along with the executable on the remote machine. This scheme, however, assumes a trusted cloud-provider/OS. Therefore, to meet our above-stated goals, 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. Roughly, the requirements for this process 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).

[5] A few SGX instructions are not implemented by OpenSGX including: Debugger Read(EDBGRD) and Debugger Write (EDBGWR) which are used to debug an SGX program. However, OpenSGX includes a GDB plug-in that performs the same task. Furthermore, some instructions are not implemented as they were not deemed necessary such as EREMOVE. EREMOVE is called to reclaim memory after an SGX program terminates, but it is not necessary as only one enclave can run at a time on top of a single OpenSGX instance.

[6] The OS emulation layer's implementation was not guided by Intel®'s SGX specification, a consequence of Intel®not specifying the interface for invoking privileged SGX instructions

[7] Simply put, the enclave program cannot invoke I/O system calls directly as any code executed by the enclave program must be part of the signed binary. System calls for I/O operations, however, delegate work to driver software which changes from machine to machine based on the hardware provider. Consequentially, allowing arbitrary system calls from within the enclave creates a large attack surface.

The first and second requirement are met by a process called inter-platform attestation, outlined by Intel® in [5] and summarized in the following section[8].

2.4.1 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 8):

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 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 (MRENCLAVE), 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.

[8] Intel® did not devise this mechanism, and it has been used before in other Trusted Execution Environments (TEE) to provision the TEE with highly sensitive secrets

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, 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.[9]

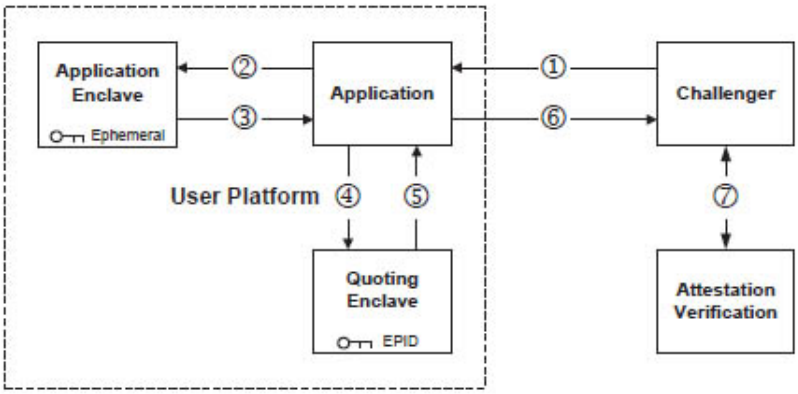

Figure 8: Remote Attestation and Secret Provisioning

Intel®has also added a construct that allows an *enclave program* to generate keys that can be used to encrypt sensitive data and persisit it on disk. The only entities privy to the key are the *enclave program* and the CPU, therefore, only these two entities can access the sealed data[10]. This process, called sealing, ensures that we execute the attestation process once, the first time the *enclave program* is launched, alleviating its cost.

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

[9] Verifying the certificate itself is conducted through contacting Intel®'s attestation verification service

[10] This is not entirely true, Intel®defined the EGETKEY instruction, the one that generates the sealing keys such that it takes an argument of either MRENCLAVE or MRSIGNER. If MRENCLAVE is used, then the sealing key generated is unique to the invoking *enclave program*. If MRSIGNER is used, however, the sealing key generated is unique to all *enclave programs* signed by that entity

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.

3 Design

3.1 The Threat Model

Before presenting our design, we introduce the threat model we utilised to scope the discussion. We consider a threat model similar to the one presented in Haven [4]. 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 compromised. 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 analysed in the Limitations section, launch a denial-of-service attack, though without compromising any secret isolation guarantees of our design.

3.2 System Design

Given the background and threat model detailed above, we may now discuss the design of our system. As previously mentioned in Section 2.1, the long-term private key is only required for the SSL/TLS handshake routine. For our design, we adopt PoLP in a manner similar to Wedge, placing the long-term private key in an SGX enclave, and defining an interface to the private key. The interface is designed to allow an SSL/TLS handshake to complete successfully, without compromising the long-term private key. The resulting, high level system architecture is depicted in Figure 9.x

Figure 9: High Level System Architecture

We assume that we have already invoked the above-detailed remote-attestation mechanism to provision the enclave with the long-term private key. The remainder of this section details the interface design for: (1) Ciphers that do not offer forward secrecy (2) Ciphers that offer forward secrecy.

3.2.1 Designing for Ciphers *Lacking* Forward Secrecy

For ease of reference, we repeat the "RSA handshake" diagram in Figure 10. The long-term private key, in this scenario, is required for decrypting $\{\texttt{PremasterSecret}\}_K$. As such, $\{\texttt{PremasterSecret}\}_K$ must be passed to the *enclave program* for decryption. However, an interface that accepts $\{\texttt{PremasterSecret}\}_K$, and returns `PremasterSecret` unencrypted compromises the long-term private key. Such an interface is an *oracle* for

the long-term private key, providing an adversary who exploits the untrusted component the ability to decrypt any cipher-text by invoking this interface.

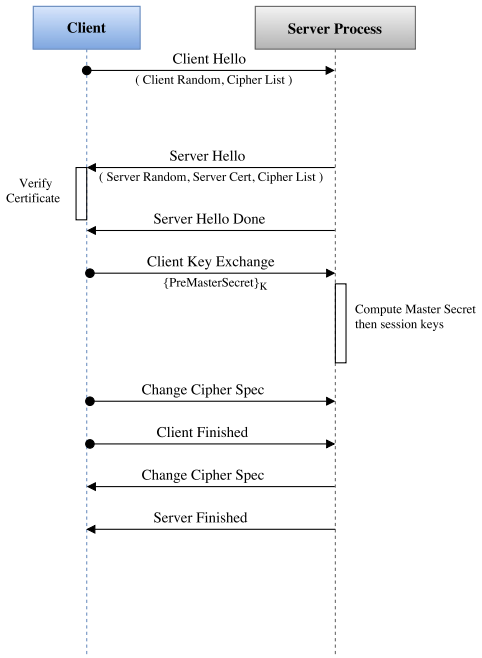

Figure 10: "RSA handshake"

Reviewing the steps for session key generation, detailed in Section 2.1, the decrypted `PremasterSecret`, along with `ClientRandom` and `ServerRandom`, is passed to a PRF to compute `MasterSecret`. Output of a PRF is *preimage resistant*, meaning that the output cannot be mapped back to the input; hence, we may choose to offer an interface that accepts `ClientRandom`, `ServerRandom`, and $\{\texttt{PremasterSecret}\}_K$, and returns `MasterSecret`. However, such an interface allows an adversary to *influence* the generation of `MasterSecret`. To be precise, the aforementioned interface allows an adversary to generate the `MasterSecret` for any eavesdropped session (`ServerRandom` and `ClientRandom` are both sent in the clear, and the interface takes $\{\texttt{PremasterSecret}\}_K$, all of these are easily captured by a passive eavesdropper).

Instead, we offer the same interface proposed in the Wedge paper, and detailed in Section 2.2.1. This interface accepts `ClientRandom` and $\{\texttt{PremasterSecret}\}_K$, both are provided by the client, and returns `MasterSecret`. `ServerRandom` is generated by the *enclave program* and used in computing `MasterSecret`. By designing the interface in this manner, we retain the freshness property discussed earlier. Consequentially, an adversary who exploits the untrusted component is no longer capable of influencing the `MasterSecret` generation routine, because one of its parameters, `ServerRandom`, is calculated by the *enclave program*. The resulting design is illustrated in Figure 11.

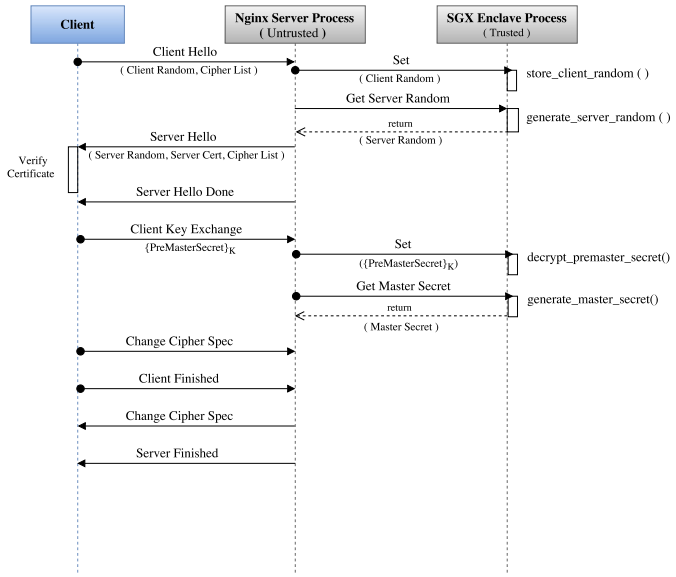

Figure 11: "RSA handshake" with private key in an SGX enclave

For clarification, we explain the "RSA handshake's" execution within the context of our design. The handshake begins with the client connecting to the server by sending a `ClientHello` message. The `ClientRandom` value contained in this message is passed to the enclave, preparing for `MasterSecret`'s derivation. After successfully feeding the enclave with `ClientRandom`, the untrusted component sends a command to the *enclave program*, requesting the generation of `ServerRandom`. Upon receiving `ServerRandom` from the *enclave program*, the server replies to the client with a `ServerHello` message, followed by a `ServerDone` message. `ServerDone` informs the client that the server is ready to proceed to the key negotiation phase. The client responds with a `ClientKeyExchange` message, containing $\{\texttt{PremasterSecret}\}_K$, where `K` is the server's public key. The server forwards $\{\texttt{PremasterSecret}\}_K$ to the *enclave program*. The code running within the enclave uses the provisioned long-term private key to decrypt $\{\texttt{PremasterSecret}\}_K$. The now-decrypted `PremasterSecret`, along with the pre-negotiated `ClientRandom` and `ServerRandom` values are input to the aforementioned PRF, outputting `MasterSecret`. The remainder of the handshake continues without any more interactions with the *enclave program* [11].

3.2.2 Designing for Ciphers *Offering* Forward Secrecy

At the time the Wedge paper was published, ciphers that offer forward secrecy were not common place; hence, the authors of Wedge deemed it unnecessary to provide support

[11] Actually, there is one more interaction with the enclave upon successful termination of an SSL handshake. This is mostly an implementation detail regarding SSL session management and Nginx's event-driven design and is not directly connected with the SSL/TLS protocol itself. We reason further about this requirement in Section 4.

for this category of ciphers. However, forward secrecy has gained popularity since, with [STATISTIC][CITATION]. We have, therefore, designed our system to accomodate ciphers within this category. As of TLSv1.2 ciphers that offer forward secrecy employ some variant of Diffie-Hellman (DH) Key Exchange to establish the `PremasterSecret`. Before discussing the SSL/TLS handsake for these ciphers, we provide a brief explaination of DH Key Exchange.

DH Key Exchange is an asymmetric cryptographic algorithm used to arrive at a shared secret between two parties. DH Key Exchange can be best illustrated using the famous "paint mixing" analogy. The analogy is presented in Figure 12.

1. The partiticpants agree to using a publicly known "paint color"
2. Each participant generates a secret "paint color"
3. Each participant "mixes" their "secret color" with the "public color" they previously agreed to use
4. The participants *exchange* the "paint mixtures" from the previous step
5. The participants combine the "paint mixture" they received with their own "secret paint". Both parties now possess the same "shared secret paint".

The security of DH Key Exchange relies on the assumption that it is computationally difficult to seperate the "paint mixtures".

Figure 12: DH Key Exchange paint analogy

Moreover, by creating new "secret paints" for each session and purging all the "paints" from memory after a session is complete, one obtains Ephemeral DH Key Exchange (DHE). DHE offers forward secrecy because no long-term private secret is used to negotiate the shared secret; hence, compromising a single session's "secret paints" does not affect any other session (new "secret paints" are used for new sessions).

For our project, we concentrated on Elliptical Curves DHE (ECDHE). This is a variant of DHE that utilises Elliptical Curves to "mix" the "paints". The mathematics behind ECDHE is beyond the scope of this report, but the interested reader may refer to [CITATION] for further details. The steps in the SSL/TLS handshake when using an ECDHE cipher ("ECDHE handshake") are presented in Figure 13.

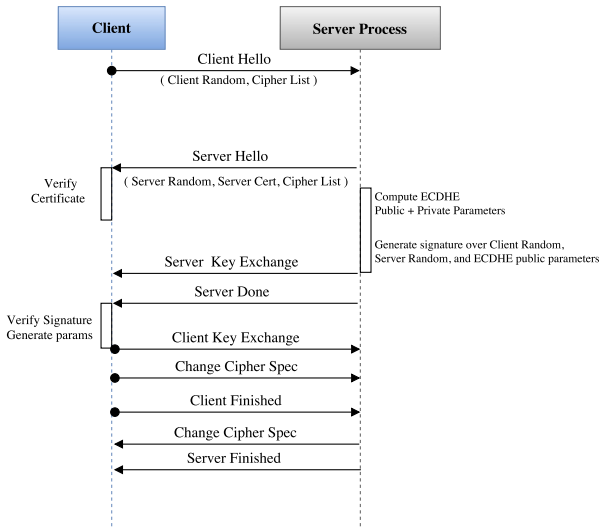

Figure 13: SSL/TLS handshake when using an ECDHE cipher

Similair to the "RSA handshake", the "ECDHE handshake" begins by establishing the values for `ClientRandom` and `ServerRandom`. In performing this exchange, the involved parties also agree on the Elliptic Curve ("well-known paint") to be used in negotiating the shared secret. Following that, the server computes its private parameter ("secret paint") and the corresponding public parameter ("paint mixture"). A hash function is applied across the public parameter, `ServerRandom`, and `ClientRandom`. This resulting message digest is signed using the long-term private key and is sent to the client, attached to the public parameter. The digital signature is computed to:

- Prove the identity of the server to the client
- Prove that the ECDHE public parameter received was indeed computed by the server, preventing man in the middle attacks.

The client verifies the signature received, computes their own private and public parameters, and sends the public parameter to the server. The client and server may now derive the `PremasterSecret`, `MasterSecret`, and the session key block. The handshake then proceeds in the same way as the "RSA handshake".

Clearly, from the discussion above, the long-term private key is used only for verification purposes in the "ECDHE handshake". To secure the long-term secret, one may elect to provide an interface that receives `ClientRandom`, `ServerRandom`, and ECDHE public parameter, computes the message digest, and derives the signature. However, such an interface allows an adversary to *influence* key generation, if they explot the untrusted component. Instead, we created an interface that accepts `ClientRandom` and returns the signature. The values of `ServerRandom` , the private parameter, and the public parameter are computed in the enclave. This provides the same freshness property described earlier in the context of the "RSA handshake", preventing the attacker

from acquiring any useful information by exploiting the interface. The resulting interface is depicted in14

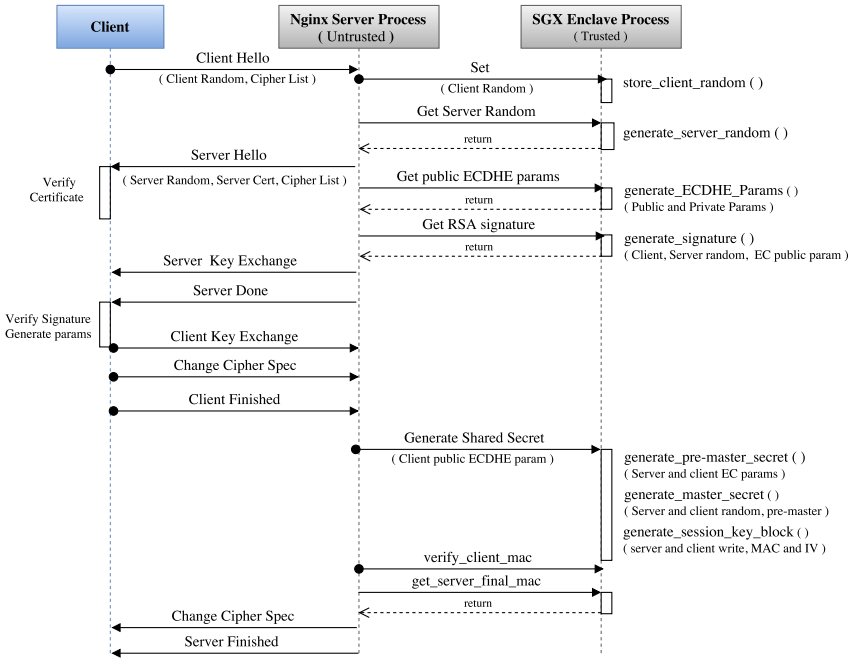

Figure 14: "ECDHE Handshake" with long-term private key in an SGX enclave

Observe, however, that `MasterSecret` is used to compute the session key block in the untrusted component; hence, an adversary, after exploiting the untrusted component, may leak the session key block (or leak `MasterSecret` and derive the session key block themselves).

3.2.3 On Isolating Session Keys

Regarding the design where we want to protect both the server's private key and the SSL session's key block, we augment the previously defined interface from the point where the $\{\texttt{PremasterSecret}\}_K$ is available to the enclave. For this scenario, we do not expose the `MasterSecret` to the untrusted component, as this would merely provide an oracle for session key derivation to an adversary. The server asks the enclave to generate the session key block using the pre-negotiated secrets and this key block does never leave the trusted component. At this point, the client signals the server with a `ChangeCipherSpec` message about transitioning to a specific CipherSpec and the server interacts with the enclave to initialize the client's cipher specific environment. The server now prepares for receiving the `ClientFinished` message by calculating the

digest of the handshake messages it received so far [12]. This step entails a dependency of the established session key, thus a context switch to the trusted environment is required for calculating the digest of the handshake buffer. Upon receiving the encrypted `ClientFinished` message, the server forwards the message intact to the enclave for decryption through the decrypt interface and calculates the MAC of the decrypted message to verify its authenticity and integrity. If this process successfully completes, the server signals the client for transitioning to a specific cipher and sets up the appropriate cipher environment in the enclave as well. Finally, it asks the enclave to calculate the digest of the handshake messages it received so far and encrypts this digest within the encalve using the established session keys. If the client successfully verifies all security properties of the `ServerFinished` message, the SSL connection is considered established. Note that, as the protocol defines, the interface used for encryption/ decryption of the `Finished` messages is the same used for encryption/ decryption of the user data travelling over the network. Thus, every subsequent data record will be encrypted/decrypted by the same interface used for encrypting/decrypting the `Finished` messages, with the session keys never being exposed to the untrusted component.

Session keys available to the OS If our sole goal is to protect the long term, private key then we may decide it is safe to keep the active sessions' keys outside of the TCB. This allows the compromised OS to use the keys as it pleases. It can just read them out from memory and send to a man in the middle (on a different to compromised machine). MITM could then decrypt previously captured traffic. This method it is possible to leak current (and cached) sessions keys. Old, captured sessions for which session keys have been deleted can no longer be recovered. However, if the OS persists to be malicious it can always monitor future (current) connections.

One could prefer this option for performance reasons - such design puts less burden on the enclave program. In particular the role of the enclave can finish after the master secret is computed from server and client randoms and pre-master secret. This is because a pseudo-random function (PRF) lies between master secret and the input values so no information can be learned about the private key. The derivation of session keys and encryption / decryption of messages can be left unaltered in the untrusted program and therefore less ćontextświtches to the enclave would be required.

Session keys hidden within the enclave One may want to further protect the key material by requiring that the session keys themselves do not leave the enclave. Instead an encrypt / decrypt oracle interface is presented to the untrusted component. The enclave performs the cryptographic operations using stored state accessed by the session_id. This prevents the compromised OS from leaking the session keys to a different machine, but requires a more complicated and expensive design. One can imagine that if the attacker is leaking ton of data out of the server, it may be more easily noticeable than if the session keys are exported outside of the server once.

[12] As defined by TLS 1.2, the digest enclosed in the `Finished` messages does not include the `Finished` messages themselves.

If we decide to keep our session keys in the enclave apart from providing a mechanism to transfer the cipher context we have to additionally present an interface that would allow to sign the handshake finish messages.

To allow the trusted component to encrypt / decrypt messages we also need to transfer the symmetric cipher context which can be done during computation of master secret. Additionally, every time we need to pass the initialization values such as initialization vector or nonce need to be transferred to the enclave.

The performance cost is increased since now at least 4 context switches between untrusted and trusted processes are required for each packet processing, i.e. 2 switches to decrypt an incoming packet and pass plain-text back to the application and 2 to encrypt the outgoing result after application processing (plain-text in, cipher-text out of an enclave).

Also every handshake now needs 6 (TODO: ???) more context switches during digest update.

4 Implementation

In this section, we discuss the various parts of the system components that we have modified during the different phases of the project. The main implementation parts comprise modification to Nginx-Webserver's SSL event handling layer , partitioning the LibreSSL engine, openSGX hardening , generate automated build setup and develop an IPC interface. This section also covers the improvements that we have made to each component to achieve our final goal. The discussion on the work carried out for testing the system, performance monitoring etc are saved for a later chapters.

The primary focus of the work was to separate all the sensitive data (particularly private keys) from the network facing component of the Web server to a secure region. To achieve this, we had to separate out the key management part of the SSL layer from the Nginx server. Figure 2 illustrates the system components and their corresponding partitions.

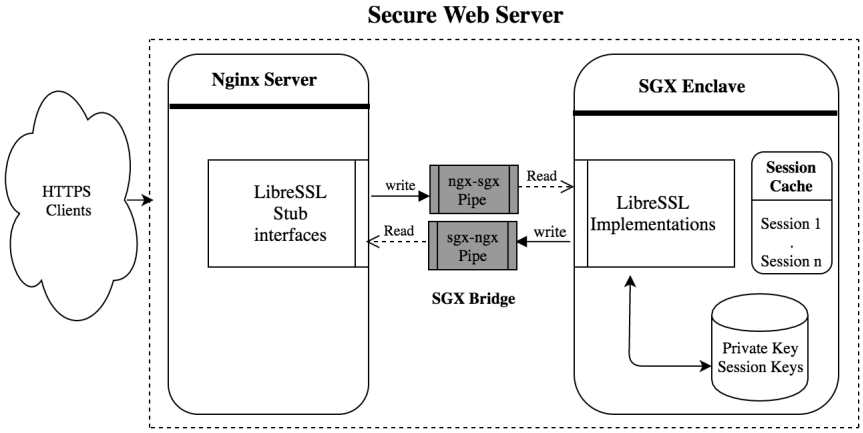

Figure 15: Secure Webserver block diagram

In the following section, we briefly discuss about the different aspects of each of the system components, the difficulties faced during the implementation phase and the rationale leading to any particular approach that we have taken.

NGINX Server . NGINX is an open source web server that uses an asynchronous event-driven approach for handling requests from the clients. The rationale behind choosing Nginx over Apache is the performance improvement it offers with the event driven architecture. In Nginx, each worker process handles thousands of connections simultaneously as opposed to Apache's process/thread per connection approach. Furthermore Nginx is lightweight, scalable and also gaining lot of market share compared to Apache.

The Nginx server process can manage multiple clients simultaneously and the request events are handled individually as they arrive. The server is capable to offload the tasks to multiple worker processes in case of heavy incoming traffic. However we have configured it to run as a single worker process. This arrangement is to make it compatible with the shortcoming of openSGX i.e. it can execute only one enclave at any instance.

In short, it supports only a single execution thread at a time. Hence regardless of how many worker processes are requesting, only one can be served by openSGX. We have modified nginx's default configuration file to facilitate this. (/etc/nginx/nginx.conf).

SSL Engine . We have used LibreSSL library to implement the SSL/TLS protocols in the system. LibreSSL is a fork of OpenSSL aiming to provide more secure implementation. LibreSSL has undergone lot of code pruning and majority of the changes went into the library were related to the security vulnerabilities found in OpenSSL.

The library is linked to both Nginx and Enclave processes. The Libressl variant attached to the Nginx process has been modified, such that the implementation of all crypto operations are replaced with IPC read/write stubs to SGX-Bridge (refer next section). This means, all the secure operations of the handshake such as signing, verification, key generation etc will invoke an IPC request to the SGX counterpart. The SGX part of LibreSSL will execute the operations and respond accordingly. We limited the SGX code only with cryptographic operations to reduce the size of the Trusted Computing Base. Most of the code base for connection establishment/termination, state machine handling etc are still maintained outside the enclave.

It is interesting to note that in the SGX part, we maintain a session cache based on a dynamic hash table (SSL - LHASH routines) and this distinguishes multiple ongoing sessions with their corresponding session ids. This empowers us not only to handle concurrent SSL handshakes but also deal with the SSL session resumption within the enclave. (More details on cache implementation.. John?)

During the first phase of development, we isolated all the private key related operations into the SGX component and returned only master key to the untrusted part of the server. To further enhance the security, we moved the symmetric session key generation as well to the SGX enclave. This brings all the cryptographic operations within the secure compartment. It's interesting to note that, after the key block generation we had to return IV/nonce and key-block length back to the untrusted part of the server. These parameters are necessary for changing cipher states in the untrusted Libressl part. (more to expand by john ??)

We used SSL AEAD (Authenticated Encryption with Associated Data) interfaces for encrypting/decrypting the application data. The AEAD operation is designed to provide both data authenticity (integrity) and confidentiality in a single step. The underlying mode of operation for the AEAD symmetric key crypto is GCM (Galois Counter Mode). GCM is highly efficient and yields better performance. Furthermore GCM achieves the higher throughput rates with reasonable hardware resources by processing instructions parallely in contrast to CBC mode which incurs significant pipeline stalls.

SGX Bridge .

SGX Bridge is a Named Pipe IPC interface developed for data exchange between Nginx and SGX enclave process. It constitutes all the functions and data structures to read to and write from the enclave.

There are two main APIs which are used to write one of the predefined command (with its associated data) to the enclave, and read the result back to the server. > sgxbridge_pipe_write_cmd(SSL *s, int cmd, int len, unsigned char* data) > sgxbridge_pipe_read(int len, unsigned char* data);

In the real SGX hardware, the enclave will exist within the same process virtual memory and can be accessed directly without aiding an IPC. We have decided to use

these interfaces for our project as it was the IPC facility readily available within the open SGX library.

Compiler, Tools and Build system . All the implementation parts were written in 'C' programming language. At various phases of the developments, we had to use tools such as Valgrind for profiling, Callgrind for call stack analysis and gdb for debugging etc. We have developed and maintained a single bash script that automates the building and installation of all the software packages used within the system.

A conditional compilation flag "OPENSSL_WITH_SGX" has been introduced in the LibreSSL engine to aid switching between native mode and sgx mode during the build process. It is also a reference point to identify sgx call gates introduced in Libressl as part of the project.

Busy Wait - (Is it worth mentioning here or in test set up/perf eval ???) .

OpenSGX - (Anything interesting that not mentioned in the previous sections ???) .

5 Performance Evaluation

We developed our prototype using OpenSGX which, as detailed in Section 2, is an emulator for the SGX instruction set and relies on QEMU to execute the trusted code. Consequentially, we cannot measure end to end performance by simply executing our prototype and tabulating how many requests per second are completed, due to the overhead of emulation. Instead, we resorted to modeling the performance of the prototype in a manner similar to [4].

The remainder of this section describes the model we implemented to carry out our performance measurements and details the results we acquired from running our tests.

5.1 Performance Model

To model the performance of our prototype we implemented a second version of our enclave program that has no dependencies on OpenSGX and replaces certain SGX instructions with busy waits, simulating the overhead in executing them. In dong so, we make the same assumptions as [4] namely:

1. We assume that the CPU used in testing performs the same as an SGX-enabled CPU for all non-SGX instructions

2. We assume that the EPC is large enough to accommodate the entirety of the trusted component and any data structures created after program start-up

Leaving the overhead of SGX instructions, memory encryption, and asynchronous exits unaccounted for.

SGX instructions that bootstrap the enclave and verify its integrity are only executed once at startup, and therefore have no effect on runtime performance. As a result, we only simulate the overhead of EENTER, ERESUME and EEXIT. Simulating the slow down in memory accesses due to the need to decrypt the RAM contents before being able to access them in processors cache, was carried out by reducing the memory's clock in [4]. Such a proxy works for their system because it executes within an enclave in its entirety. Clearly, such an approach would underestimate the performance of our system because only the trusted component incurs the overhead of memory encryption. Instead we captured the number of Last Layer Cache (LLC) misses as a relevant measure.

Finally, the performance of 'malloc' is slower for our test program than what it would be within real SGX because we have to context switch to the OS to allocate memory.

5.2 Hypothesis

We expect to see the most performance slowdown due to expensive context switches between untrusted and enclave programs. This is due to the required TLB flush (??) accompanying EENTER and EEXIT instructions.

For the case when master secret and session keys are available to the OS we don't expect significant slowdown as it only adds 4 context switches during the handshake that can be amortised over multiple connections.

In contrast, the second case, providing stronger security guarantees, should perform considerably poorer due to extra XXX context switches during the handshake. Moreover this scenario incurs additional slowdown due to the extra 4 context switches for every request made to the server.

vary the response size for the keys outside too - to verify our hypothesis that its not going to be performance drop

5.3 Microbenchmarks

In order to gain better insight about our implementation we measured its following aspects.

OpenSGX can report the performance measurement of an enclave program which includes the number of kernel switches and number and type of enclave instructions executed. We report the relevant number of instructions per request.

To be able to reason about performance hit due to memory decryption we captured the number of LLC misses using Intel VTune Amplifier. We tried to verify its reported values using perf tool, but chose the former because it had all CPU specific counters predefined in the program and was available under student license. perf also lacked thourough documentation and would require consulting Intel Reference Manual for the counter values for our CPU.

To measure the size of the TCB of the enclave program we generated its static call graph (using Egypt) to determine all the libressl entry point functions and how they follow internally. We then manually copied the source of all functions used and counted the resulting SLOC (with cloc). We also report the number of changes we have made to the libressl and nginx source code.

5.4 Tests setup

We run our tests on a Amazon EC2 XXX instances, with the server and client located on different virtual machines within the same data center. !!!TODO: needs updating with what we actually end up using!!! These machines have 1 virtual CPU, with 3.75 GiB RAM, 4 GB SSD storage, and XXX network. The underlying hardware is Intel Xeon E5-2670 v2 (or Sandy Bridge) 2.5 GHz (2.6 GHz for Sandy Bridge) [7]. We use Ubuntu 14.04 with 4.4.0-34 generic kernel for our OS and a custom script using *httperf* to generate the load and measure the performance.

We have measured the end to end performance for ciphersuites not offering forward secrecy based on RSA handshake as well as suites providing forward secrecy based on ECDHE:

1. (RSA-)AES256-GCM-SHA384
2. ECDHE-RSA-AES256-GCM-SHA384

We use the AES as it can benefit from hardware acceleration using AESNI instructions present in modern devices.

We tested two versions of our program, for the two threat models discussed in Section 3:

1. session keys available to the OS

2. session keys hidden from the OS

We have tested each of the above setups with a range of static document sizes (1 KB - 10 MB) and a range of values for the busywait instruction delays (10, 30, 50 K) - a subset of values used by [4])

We performed the test for the busywait instrumented enclave program with sgx instrumented libressl 2.4.1 statically linked to nginx 1.11.1. We ran the same set of tests with an unmodified version of nginx+libressl for baseline comparison.

We also run the test inside OpenSGX to learn which and how many SGX instructions were executed and to be able to verify our performance results.

We also varied the ratio betewen new and reused (cached) sessions.

5.5 Results

Table XXX presents the number of modifications to the source code. The enclave program TCB is XXX of sloc.

We used the formula from [page 30 in `https://software.intel.com/sites/landingpage/legacy/pdfs/Using_Intel_VTune_Amplifier_XE_on_2nd_Gen_Intel_Core_Family.pdf`] to convert the number of LLC misses to the percentage of cycles the enclave program spent on waiting for cache misses. We multiply this by the memory overhead reported in [CITATION NEEDED] (5-14 %)

Additionally we report that this not a performance bottleneck as it is under 0.2% which Intel considers ‘normal case‘.

5.6 Discussion

6 Project Management

Project Management. Outline of how the work was split up. How collaborative work was planned. Who did what. How agreement was reached. How essential project information was managed and communicated. How integration and testing was organised.

At the begining of the project we have spent about 2 weeks making sure we understand the problem and our approach to the solution. We researched software possibilities, available documentation, related work until we identified three main parts of the project:

- Secret provinsioning to the enclave through reomte attestation
- Libressl and nginx instrumentation
- Performance evaluation

Therefore, we divided our group into three subgroups:

- Ahmed and John working on the SGX part
- Betran and Wiktor working on Libressl and nginx
- Chiraphat working on the test setup and performance evaluation

We have chosen messaging app Slack as our daily communication channel since we already used it during the course duration. We setup github repositories to collaborate on the code. We also decided to use LaTex for documentation of our work and writeup of group report to follow research best practices.

Every week we met with Brad, our supeervisor, to report on our progress, make sure we are on track with our schedule and discuss any issues and tasks to be carried in the following week. Additionally, we held our own meeting to internally share what we have learned and (re)distribute further work.

We also assessed whether some tasks need more man power, which for most of the time was not necessary, until towards the end of the project when we stumbled upon more challenging implementation issues or needed to integrate parts of the project to be ready for final preformance evaluation. During that time we held more in person meetings and held pair debugging sessions. These turned out to be very useful and we managed to solve the issues on time.

7 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] C. Latze and U. Ultes-Nitsche. *Transport Layer Security (TLS) Extensions for the Trusted Platform Module (TPM)*. RFC Draft. University of Fribourg, 2010. URL: https://tools.ietf.org/html/draft-latze-tls-tpm-extns-02.

[4] 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: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[5] Intel Corporation. *Intel® 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.

[6] Prerit Jain et al. "OpenSGX: An Open Platform for SGX Research". In: *Proceedings of the Network and Distributed System Security Symposium*. San Diego, CA, Feb. 2016.

[7] *Amazon EC2 Instance Types Kernel Description*. https://aws.amazon.com/ec2/instance-types/. Accessed: 2016-08-25.