0.1 The Principle Of Least Privilege

As previously stated 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 have 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. First, we list the attacks possible against a monolithic web server and then detail how the design implemented with Wedge resolved these vulnerabilities.

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. A consequence of the web server's user-level processes (dynamic content generation scripts, databases &c.) requiring plain- text to complete their tasks, thus, even if we secure the encrypt/decrypt interface using the solution in the Wedge paper, the data would be available in the non-privileged process's memory. This is further discussed in Section **??**.

Possible Attacks

There are two main attacks that may be mounted by an adversary capable of exploiting *only* the network facing component 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 component, which has to run as root to bind to port 80. This could be by reading the private key from disk directly or from the process's memory space. The attack is illustrated in Figure 1.

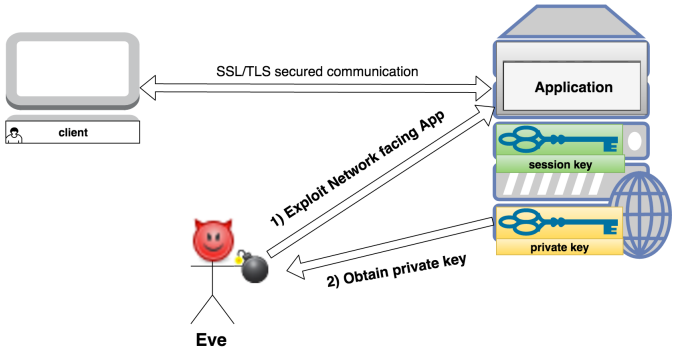

Figure 1: Exploiting the network facing component to leak the private key

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 (assuming a cipher that does not provide the property of perfect forward secrecy) 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 is one that does not offer perfect forward secrecy (PFS), and it only affects the single session which was eavesdropped. The attack is illustrated in Figure 2.

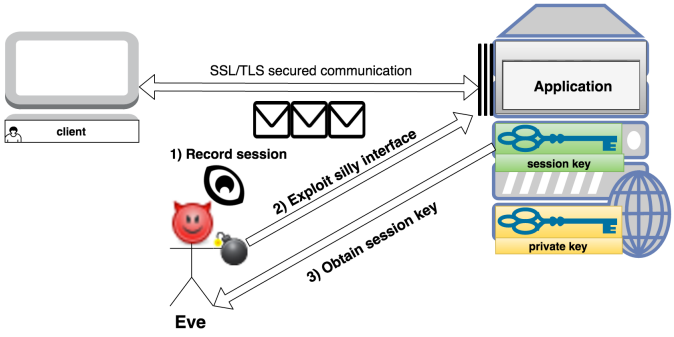

Figure 2: Exploiting the network facing component and the naive session key generation interface to generate session keys for eavesdropped session

Proposed Solution

The solution proposed in the Wedge paper is through partitioning the session key generation code into its own logical compartment [1] that executes at high privilege. This partitioning can be seen in Figure 3.

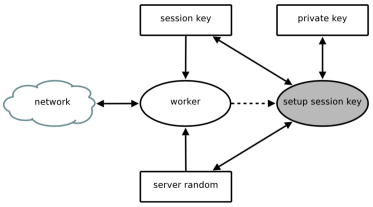

Figure 3: Partitioning Scheme to protect against leaking the private key [1]

The low privilege worker process does not have possess to 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 operating system, and may, therefore, leak the key from the privileged component's memory space, or from a malicious cloud-provider with full access to the physical machine and is capable of reading the key from disk [2]. To secure the key against the aforementioned adversaries, we utilized SGX.

References

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

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

[2] Even if the key is encrypted on disk, the decryption key must be present in the plain-text binary file for the privileged component or in the process's memory, at runtime, in plain-text