SOUL: Secure OCalls Using Library Verification

Andrew Park
Department of Electrical & Computer Engineering
Rutgers University
ap1614@rutgers.edu

I. Introduction

Intel SGX enables software to create enclaves, which are protected sections of the processor and memory which can not be accessed from the outside. Due to restrictions in memory size, it is not possible to keep whole libraries within the enclave without expanding the memory limit, which only some operating systems support. Additionally, OCalls(outside calls) can not be trusted as it runs code which lies outside of the enclave. Due to these limitations, there is no secure way for enclaves to include large libraries or use OCalls in secure manner.

SOUL aims to remove these restrictions by providing a way for the enclave to verify the code for a library before performing any OCalls. The verification is done by registering library functions into the enclave, which stores the hash of the function into memory. Before an OCall is performed, it first verifies the hash of the function before actually performing the call itself. This check creates assurance for the enclave that library functions have not been modified during execution.

Github code repository: https://github.com/Iznoanygod/secure-ocalls

II. Background

Ensuring that program runs securely is needed for many applications. DRM schemes, anti-malware, and many other similar software are subject to many attacks, such as DLL injection, which may compromise the software by allowing an attacker to hook functions and modify the code which these applications may rely on. In many cases, anti-debugging and tamper-proofing techniques are employed to prevent such attacks from occuring. However, these techniques are run on untrusted regions of memory and are not perfect solutions. An attacker is able to reverse-engineer these systems and modify the software which is running to bypass detection. TEEs such as Intel SGX allow for systems to create enclaves which are secure from attacks and allows for confidential computing that can not be modified. While enclaves do provide integrity of the software inside, it creates a situation where only the enclave can be trusted and all code outside of this enclave to be untrusted. This is problematic when code which is outside the enclave must be used, such as when a library function is called. This can be solved by storing the needed library or functions inside of the enclave itself, but this is limited by the size of the enclave. In the case of SGX, the maximum enclave size without paging is 128MB. This is problematic for software such as anti-malware, where the actual detection algorithm which needs to remain secure may be larger than this maximum enclave size. There is no solution for this situation; the enclave is not able to store the entirety of the detection software and keeping this code outside of the enclave leaves it vulnerable to attack. This creates a need for enclaves to be able to perform OCalls without risking integrity of the result. SOUL aims to provide enclaves with this assurance by creating a system where untrusted code can be verified and then run with the assurance that it has not been modified.

III. Design

SOUL uses a check-and-run system to ensure that library functions have not been modified. This is done in the following steps

A. Enclave Loading

When the program initiates, the enclave is first created and loaded from the enclave file. The enclave then performs remote attestation with a remote server to verify that the software running inside the enclave has not been modified.

B. Library Retrieval

Once the software running inside the enclave has been verified, it sends a request to the remote server for the library. The library is unique for each request and uses a random number generator to create a unique challenge-response to ensure that an attacker is not using a fake library. The library is then loaded into memory and all symbols are imported. The needed functions are registered into the enclave, which stores the hash of the function within the enclave memory to ensure it can not be modified.

C. Function Verification

When an OCall is performed, the function which is being called is first verified before it is executed. This is done by performing a hash of the function, which is then compared to the recorded hash value inside the enclave. If the function hashes match, the enclave can assume that the contents of the library have not been modified. Otherwise, mismatching hashes would indicate that the library has been modified and the enclave can not proceed with the OCall.

IV. Implementation

The repository contains sample programs, which demonstrate the different components of SOUL. Each of the sample programs build upon one component of SOUL, which is then put together to create the full system.

A. BasicLibraryLoad

`BasicLibraryLoad` loads a shared library file dynamically and its symbols table. This is done using `dlopen` and `dlsym`. `dlopen` first loads a shared object file into memory. `dlsym` then loads symbols from the loaded shared object. The shared object file we use is `library.so`, which contains a single function `dynamic_function`. The function is then called through a function pointer.

B. LibraryServer

`LibraryServer` provides a shared object to a client, which is then loaded and used by the client. The client sends a request to the server, which then the server responds by sending the full shared library. This library is then loaded using `dlopen` and the function is laoded using `dlsym`. The library used is the same as in `BasicLibraryLoad`.

C. LibraryModificationDetection

`LibraryModificationDetection` modifies the function in a library and tries to detect this modification. This is done by loading the library using `dlopen` and loading the function through `dlsym`. The hash of the function is then stored for later comparison. The memory page permissions is then changed to enable reading and writing. The program then overwrites the machine code in the library function to change the output of the library functions. This type of modification is known as self-modifying code. It tests both the function before and after modification to show the changes. Then, the function `check_modification` checks the hash of the function against the original hash that was recorded and prints out the results.

D. BasicEnclave

`BasicEnclave` creates a sample enclave with a single ECall available which multiplies two given integers together. The enclave is compiled and signed using a randomly generated RSA key. This is then loaded by the app, where the enclave function is then run using two sample numbers. The enclave itself remains secure and any computations performed inside are hidden from the app. This ensures that the enclave function can not be tampered.

E. SOUL

SOUL was implemented in a proof-of-concept built for Ubuntu 20.04. The program is in two parts, a client and a server, which communicate with each other to perform remote attestation, library retrieval, and verification.

The client is separated into two regions: the untrusted app and the trusted enclave. The untrusted app first loads the enclave file and requests a unique library from the remote server. Once this is done, the library file is loaded using `dlopen`. The challenge-response system is then used to verify that the library has not been modified by an attacker during transit and the remote server checks the response of the library. Once this is completed, the function registration process begins by requesting the function hash values from the remote server and storing these values inside of the enclave. This completes the library verification state of SOUL. When an OCall is performed, SOUL first computes the hash of the function and then checks the computed hash with the stored hash. If the hashes match, the OCall can proceed and the outcome is trusted. If the hashes do not match, this indicates that the library has been modified prior to the OCall and the call is aborted.

The server is needed for supplying a library which contains a challenge-response that is randomly generated to ensure that the library has not been modified during transit. The server then verifies the function hashes from the client enclave to ensure that the library has not been modified before function registration. Once this has been completed, the client enclave can disconnect from the server and can assume that the library that is currently loaded has not been modified.

V. Evaluation

SOUL is tested against the same function modification which was used in `LibraryModificationDetection`. When tested against self-modifying code, SOUL was able to detect that the library was modified and prevented the function from executing. Similar results are seen when testing against other common attack vectors, such as library hooking and DLL injection. In most cases, SOUL is able to detect that the library has been modified and prevented the function from being run.

SOUL did have observable increases in run-time, but this is negligible when compared to the time taken to perform the OCall itself. This increase in run-time can be attributed to the computation of the hash of the library function, which varies based on the the size of the function which is being checked. There is also an increase in run-time at the beginning of the execution of our program due to the registering of library functions into our enclave, which requires communication with a remote server and multiple OCalls.

Despite these observable run-time increases, the use of SOUL detected library modification and prevented the enclave from performing OCalls of modified library function.

VI. Limitations

The intention behind SOUL is to create assurance that OCalls are not compromised, that is to say that library functions have not been modified through the use of DLL injection, self-modifying code, or other similar types of attacks. However, due to the nature of the way OCalls are performed, it is simply not possible for OCalls to remain completely secure. SOUL does not guarantee that OCalls are completely secure, it instead creates reasonable assurance that these calls are secure. A dedicated attacker would still be able to modify the library functions and bypass the SOUL detection. This can be done through a number of means.

A. Denial-of-Service Attack

The enclave uses a remote server to make multiple different requests. Initially, the enclave performs remote attestation to ensure that the software within the enclave is not modified.

Once this is completed, the enclave requests from the server a unique library file and symbols data which contains all functions and their hashes. Due to this reliance on a remote server, an attacker can launch a Denial-of-Service attack to prevent the enclave from communicating with the remote server, effectively preventing the enclave from running and breaking SOUL [1]. This can be slightly mitigated by keeping library files saved locally on the client machine, but this enables attackers to analyze the library and devise more targeted attacks.

B. Timing Attack

Because of the method which SOUL checks the library functions, there exists a timing attack which may compromise the integrity of the library. This is due to the ordering of the check and run. A skilled attacker would be able to place a breakpoint at any moment after the function check is performed and before the OCall is made. This type of attack can be prevented using standard anti-debugging techniques, which may include self-debugging, time-based detection, and TLS callbacks [2]. While this does not prevent an attack, it does make it more difficult to perform these attacks. when paired with standard anti-debugging and other similar techniques, such as those used in anti-cheat software, it is almost impossible for an attacker to compromise the system.

C. Spectre-Style Attack

Spectre-style attacks are able to leak information from the enclave such as secure data and even attestation keys [3] [4]. This can compromise the integrity of the enclave itself, which would break SOUL. Despite mitigations implemented to prevent these types of attacks, the vulnerabilities still exist and can not be avoided without new patches which remove these side-channel vulnerabilities. As such, it is not within SOULs ability to prevent Spectre-style attacks.

VII. Further Work

Further work can be done to improve the security and performance of SOUL. The implementation of SOUL does not use remote attestation, although the design calls for this step as part of the enclave loading process. This does not change the integrity of the OCalls, but does verify the integrity of the enclave itself, which may be subject to attack. Additionally, implementing standard anti-debugging and tamper-proofing techniques would increase the security of the untrusted portion of the program. Combining SOUL with standard security techniques will increase the assurance that the untrusted region of code has not been modified by a malicious attacker, and if it has been modified, it creates a way for the enclave to detect and flag these functions as having been modified.

VIII. Conclusion

SOUL is able to provide enclaves reasonable assurance that OCalls performed have not been modified and that the result of these inherently unsafe functions can be trusted. This is done by registering functions into the enclave by hashing the functions contents and checking the contents of the function when an OCall is performed. This aims to prevent library functions from being modified, which prevents common attack vectors such as DLL injection or self-modifying code.

References

[1] Y. Jang, J. Lee, S. Lee, and T. Kim, "Sgx-bomb: Locking down the processor via rowhammer attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3152701.3152709

[2] A. K. Oleg Kulchytskyy, "Anti debugging protection techniques with examples," *apriorit*, 2021. [Online]. Available: https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software

[3] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxeattack.com/, 2020.

[4] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," 2020.

[5] Huron. (2019) Deploying intel sgx applications. [Online]. Available: https://hurondocs.readthedocs.io/en/latest/intel-sgx-deployment.html

[6] S. S. . S. Lab. (2019) Sgx 101. [Online]. Available: https://sgx101.gitbook.io/sgx101/

[7] Intel. (2021) Intel(r) software guard extensions for linux* os. [Online]. Available: https://github.com/intel/linux-sgx

[8] D. Roy. (2011) Let's hook a library function. [Online]. Available: https://www.opensourceforu.com/2011/08/lets-hook-a-library-function/