Epuri - Sandbox Programs from Dynamic Linker Hijacking via Seccomp Filtering

https://github.com/ctkhanhly/epuri

Ly Cao
Department of Computer Science
New York University
lc3940@nyu.edu

***Abstract*—This paper presents the dynamic linker hijacking problem where malicious code can be injected and override the main program or any linked shared library function by linking a malicious dynamic library to the main program via LD_PRELOAD environment variable. As all libraries listed in LD_PRELOAD are loaded first and can override any function in shared libraries with the same name, the attacker can manipulate the main program in a dangerous way. The paper proposes a sand-boxing library libsimplenoti that utilizes seccomp filtering via libseccomp. Seccomp filtering allows users to prevent malicious system calls and their arguments by actions such as notifying a tracer process, logging, returning an error, or killing the main program on the specified system call that needs to be filtered [1]. Furthermore, seccomp notify can be extended to intercept system calls and inspect their arguments, only allowing intended calls to proceed and returning an error otherwise. Libseccomp is loaded as the last library in LD_PRELOAD, so the main program is protected with seccomp before any malicious library will be loaded, making sand-boxing the main program possible.**

***Index Terms*—dynamic linker hijacking, LD_PRELOAD, shared libraries, seccomp filter, sandbox, openssl**

I. Introduction

Dynamic linker hijacking is an attack that overrides shared library code by compiling a modified library and adding it to LD_PRELOAD environment variable on Linux and DYLD_INSERT_LIBRARIES on MacOS [2]. The libraries are loaded before other shared libraries of a program and can override functions defined in the original libraries. This is dangerous as the attacker can inject malicious code into commonly used shared libraries such as openssl (libssl) and expose user's credentials. Libssl provides cryptographic functions that are commonly used in secure networking protocols such as Transport Layer Protocol (TLS), a secure network protocol, or Secure Socket Shell (SSH) which allows a user to gain access to a remote computer on the internet. User's credentials include a private and public key where the public key is available to everyone and only the user has private key. Web servers also require certificates that verify their credentials and a list of certificate authorities that can validate these certificates. With the access to these credentials, attacker can log into the host's system. Eventually, the host system is at risk and is vulnerable to more attacks. Seccomp filtering provides a way to protect programs against these attacks by allowing user to specify which system calls and their specific arguments are allowed and provides various mechanisms to inspect unexpected system calls such as notifying another tracer process, logging, or killing the running process. In this project, I am using libseccomp library to implement seccomp filtering programs as it wraps simple Application Programming Interface (API) calls over Berkeley Packet Filter (BPF) filtering syntaxes which is then applied to Linux's seccomp filtering APIs.

To understand the impact of a dynamic linker hijacking attack, I develop a model on how an attack can happen and provide a sand-boxing solution to prevent the attack. In the attack model, some malicious hacker has modified some openssl's functions and compiles them to a library called fssl so that when functions that read private key, certificate, or certificate authority are called, the attacker will send the content of these files to their server that has been waiting to get the information. The affected program is a redis client which makes a mutual TLS (mTLS) connection to a redis server. Mutual TLS is a special protocol that requires authentication from both the client and server connecting securely using TLS instead of the standard TLS which only requires server's authentication [3]. Since redis uses openssl for their TLS implementation, my attack model can successfully intercept openssl calls and steal the client program's credentials.

As a solution, I create a sandbox library called libsimplenoti that opens two processes per program, one is a tracer process that has a full capability and monitors malicious system calls from the main program, which is protected by a seccomp filtering that notifies the tracer on very call to the "connect" system call. The tracer process will check whether the destination of the "connect" system call is the redis server and only allows outgoing connection to the redis server. To prevent Time Of Check To Time Of Use (TOCTTOU) attack [4] where another thread modifies a shared argument after the "connect" call was intercepted and passed the tracer's process check, system calls "fork", "vfork", and "clone" are not allowed and will lead to the main program being killed if it attempts to use any of them. Thus, libsimplenoti is only suitable for single threaded programs.

There are four models to test against the effectiveness of my sand-boxing library and the investigation of the flaw in

the TLS implementation of redis client. Prior to running redis client program, a script is used to generate server's, client's, and certificate authority's private key and certificate containing information in the configuration such as Internet Protocol (IP) address and the corresponding public key. The configuration used for server's key generation will also include the server's IP address. There are two configurations used in the client program. Configuration 1 (config 1) for the client contains the client's IP address while configuration 2 (config 2) for the client doesn't. The configuration for a self-served certificate authority will not include any IP address. In the first model, redis client is run with LD_PRELOAD set to the malicious shared library called libfssl where the attacker overrides some functions with the same names in openssl using config 1. The second model is the same as the first model with the exception that it uses config 2. In the the third model, redis client is run with LD_PRELOAD set to libfssl and libsimplenoti using config 2. In the fourth model, the attacker will use the file contents which are private key, certificate, and certificate authority obtained from either the first or the second model and connect to redis server.

The paper will be divided into 9 sections: introduction, motivating example, hypothesis, Epuri overview, empirical evidence, other secure practices, performance, related research, followed by the conclusion and future work. Section 1, introduction, gives an overview of the project and the experiments used to test the hypotheses. Section 2, motivating example, talks about Ebury, an attack using dynamic linker hijacking that inspires Epuri. Section 3 shows the hypotheses on the three attack models proposed in the introduction. Section 4, Epuri overview, talks about the overall prototype system that the three models are run on and why I choose to develop my models as they are and the hypotheses I have for each. In section 5, empirical evidence, I will discuss why the third model is secure and the first and second model are not and conclude on whether my sandbox solution is successful. In the sixth section, I will discuss what some other secure practices are that should be used to prevent dynamic hijacking attacks using LD_PRELOAD environment variable. In the seventh section, I will discuss a performance measurement I made on the overhead of using libsimplenoti. Section 8 on related research references other papers that are relevant to Epuri. The conclusion in section 9 wraps up the project by summarizing the experiments and their results, as well as clarifying on future work that could improve the current solution.

II. Motivating example

Ebury is an attack on openssl that modifies libssl and loads the modified library on user's application using LD_PRELOAD. In this project, I want to replicate how an attacker might inject malicious code using dynamic linker hijacking similar to Ebury and provide a solution to protect programs against such attacks. All software will be protected using a sand-boxing library that's pre-loaded before the program execution. User passes in the system calls that need to be monitored and has the ability to monitor them in the tracer process that only allows system calls and arguments that are expected. The sandbox library will make sure that all requested system calls only proceed as expected. As a result, it prevents the injected malware from performing any malicious action.

III. Hypothesis

In the first model, redis client is pre-loaded with libfssl using config 1. Even though redis client is attacked and the attacker can retrieve the the victim's credentials, the redis client program on the attacker's machine should not be successful in making a connection with redis server as the attacker's IP address is different from the victim's IP address, which redis client should have checked prior to claiming that the attacker's identity is verified.

In the second model, the IP address is eliminated from config 2, so the attacker should be able to connect to redis server using the obtained credentials from the victim's machine.

In the third model, libsimplenoti is appended to the end of the list of pre-loaded libraries defined in LD_PRELOAD. In Epuri, the only 2 pre-loaded libraries are the malware (libfssl) and the sandbox (libsimplenoti), so LD_PRELOAD="libfssl.so libsimplenoti.so." Since LD_PRELOAD loads libraries in the reverse order [5], we make sure that libfssl cannot open the connection to the attacker via connect system call as it is subject under seccomp filtering rules, which only allows outgoing connection to redis server. Thus, in this model, the attacker cannot obtain the victim's credentials to run redis client program with; hence, they cannot connect to redis server.

The fourth model will be run after the first and the second model following the check that the attacker could receive victim's credentials. On the other hand, in the third model, we only need to verify that the attacker could not receive the victim's credentials.

IV. Epuri Overview

Epuri consists of 4 libraries: libsimplenoti, libfssl, libsimplenet, libsimplelogger, and the main program epuri which can create a simplenet server or client based on given command line arguments. Libsimplenoti is the sandboxing library that creates a new child process and sets up seccomp filtering in the child process on load and lets the parent process become the tracer. Libsimplenoti overrides libc main function so that only the child process that is seccomp-protected will run user program that this library is linked to. Libfssl is the malware library which overrides SSL_CTX_use_PrivateKey_file, SSL_CTX_use_certificate_chain_file, and SSL_CTX_load_verify_locations which each gets the private key, certificate, and certificate authority file name respectively, reads the file, and sends the contents over the network to the attacker's server. Upon loading, libfssl will make a connection to the attacker and in every overridden function, it sends back the credentials to the attacker via this connection. Libsimplenet is a simple networking library that libfssl uses

to make network calls and provides APIs to set up server and client programs. Libsimplenet assumes only one client ever connects to the malicious server and sends one of four types of message: PRIVATE_KEY_MESSAGE, CERT_MESSAGE, CACERT_MESSAGE, CLOSE_MESSAGE which receives private key, certificate, certificate authority file contents, or close the connection respectively. Libsimplelogger is a library that logs based on the specified file path instead of printing out logs on standard output (stdout). This is useful to debug and prevent printing out any error message from the malware on the host's main program. The setup of the program is such that attacker, server, and the victim will be on the same virtual network created by docker containers. Ideally, the setup should be such that the attacker, server, and the victim are on different machines, and the attacker is listening on one port while the server is listening on another port on their respective machines, with nat forwarding set up properly so that the victim can reach both machines. However, this is overkill to test the hypotheses I had for each model and makes epuri less accessible for anyone to try out. The attacker has an IP address of 10.9.0.2 and is listening on port 8346 while the server is running redis server listening on 8345 with the IP address of 10.9.0.3. The victim is running a redis client program trying to connect to redis server on port 8345 with an IP address of 10.9.0.4.

V. Empirical Evidence

The experiments show that the second and third model behave as expected while the first model doesn't. In the second and first model, the attacker is able to retrieve private key, certificate, and certificate authority files and use them to run a redis client and successfully connect to redis server. However, in the first model, we expect that the the attacker cannot use redis client and the victim's credentials to connect to the redis server as the attacker and victim have different IP addresses. As a result, this shows a flaw in the TLS implementation of redis client. In the third model, the attacker was not able to retrieve the victim's credential files, which is expected.

VI. Other Secure Practices

Seccomp filtering provides a mechanism to build a sandbox environment to run any application code, which allows inspecting system calls, killing, logging, and monitoring programs when a malicious system call is encountered. As a rule of thumb, there are other security practices when it comes to using a shared library such as checking the official hash of the library against the hash of the version on your machine. If possible, only whitelist intended system calls and their arguments and disallow all other system calls. Use libraries that check for the program's control flow logic to make sure the code doesn't execute injected code in the heap or in the stack. Last but not least, make certain memory and files as read-only (such as some internal data structure via "mprotect" [4]) and separate data and executable regions in the program.

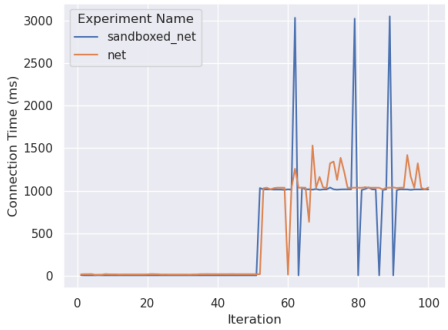

Fig. 1. Time (milliseconds) it takes for a client to connect to server with libsimplenet versus without

VII. Performance

There is an overhead of around 1.09 milliseconds per connection when the program is sand-boxed using libsimplenoti. This overhead needs to be considered for applications that require high performance networking capabilities. The performance script was tested on Ubuntu 20.04 on an Intel(R) Core(TM) i7-9700KF CPU running at 3.60GHz, 16GB RAM. To measure the impact of libsimplenoti on the latency of each connect system call, I ran two experiments. Sanboxed_net makes connections from a client to the server 100 times and records the time it takes for each connection with the interference of libsimplenoti. On the other hand, net also records the connection time 100 times but doesn't use libsimplenoti. From figure 1, we notice that from iterations 1 to 50, the connection bottleneck is similar between a sand-boxed and an unsand-boxed version, but starts to diverge from iterations 51 to 100. It is not clear why some calls are slower in sanboxed_net and some are slower in net, so the true latency of sanboxed_net could be less. The spikes in 1 could have been due to the nature of the network system calls.

VIII. Related Research

DeMarinis et al. [6] develops a system that constructs function call graph for any given binary program and compiles a seccomp filtering program as a shared library, denying any system call that is not included in the constructed function call graph. This design follows the principle of least privilege which is a recommended way of limiting only needed system calls and their expected arguments, but is limited in how an invalid system call is handled. Taking the arguments of connect system call is a file descriptor number, an address to a sock_addr structure, and the length of that structure as an example. If we purely restrict "connect" on the address number and file descriptor, must open the socket file descriptor and pre-allocate a read-only sock_addr structure, which makes it more difficult to use "sysfilter" [6] for any binary program. In the solution I'm proposing, we can simply build a custom sandbox program that notifies a tracer on the system call requested to be inspected, and get the data in the the sock_addr structure from protected program's address space, and decide

whether we should allow the kernel to execute the system call or not. The solution uses seccomp notify feature [7], which was developed in Linux kernel 5.0 March 3, 2019 and improved in Linux Kernel 5.9 on October 11, 2020 [8].

Payer et al. [4] builds a prototype sand-boxing system called libdetox similar to seccomp in user mode whereas seccomp restricts user programs in kernel mode. Libdetox is also loaded and analyzes user's program via LD_PRELOAD. The system first translates the binary program in machine code using a dynamic binary translator, which filters out unintended system calls or arguments based on user's input policies. The translated program is then converted back into a machine code. Each thread in the program has a separate sandbox, which prevents TOCTTOU attacks. Libdetox provides more protection than libsimplenoti, but libdetox has 12.06 % overhead on average, while libsimplenoti only incurs 0.210% overhead.

Scholz et al. [9] discusses an application of BPF filtering syntax that allows filtering packets and applies the filter in the application. However, implementing BPF filtering is complex and not intuitive for general programmers as it uses instructions similar to assembly language. In this project, I use libseccomp which provides API calls and generate BPF code from the wrapped seccomp filters.

IX. Conclusion and Future Work

In conclusion, the paper not only explores how a dynamic linker hijacking attack can be made possible, but provides a sand-boxing solution against it via libsimplenoti. In addition, the paper uses a real-world software redis client and investigates a bug in the mTLS implementation of redis, which doesn't check for client's details such as whether the incoming packet's IP address matches their IP address as specified in the certificate, which allows attack such as model 1 to run successfully. As for future work, libsimplenet could be improved that provides API calls on accepting a user filter, registering notification file descriptors via an event-based API and an API that allows user to customize methods for each notification event in the tracer process.

References

[1] Michael Kerrisk, *seccomp(2) — Linux manual page*. Linux Foundation, Vienna, Austria, August 2021.

[2] "Hijack execution flow: Dynamic linker hijacking," March 2020.

[3] "What is mutual tls (mtls)?."

[4] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, (New York, NY, USA), p. 157–168, Association for Computing Machinery, 2011.

[5] A. Sawula, "Order of ld_preload library initialization," December 2014.

[6] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, (San Sebastian), pp. 459–474, USENIX Association, Oct. 2020.

[7] C. Brauner, "Seccomp notify – new frontiers in unprivileged container development," July 2020.

[8] A. Crequy, "Seccomp notify on kubernetes," February 2021.

[9] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *30th International Teletraffic Congress (ITC 30)*, (Vienna, Austria), 2018.