Linux rootkits explained – Part 1: Dynamic linker hijacking

wiz.io/blog/linux-rootkits-explained-part-1-dynamic-linker-hijacking

Contents

Linux dynamic linker

- LD_PRELOAD
    - Difference between LD_PRELOAD and /etc/ld.so.preload
    - Usage in the wild
    - Detecting LD_PRELOAD abuse

Summary

Rootkits are a type of malware used by threat actors to gain complete control over a compromised resource and hide malicious activity. They are often part of a broader attack campaign such as stealing sensitive information or conducting espionage. Rootkits can be difficult to detect and remove since they leverage advanced techniques to conceal their presence on a system.

To achieve different capabilities, rootkits intercept and alter normal execution flow. The interception can be in different layers of the operating system, including userland-level code and kernel-level system calls.

In this blog post series, we will focus on Linux because it is the predominant operating system in the cloud. We will cover three different Linux rootkit techniques: dynamic linker hijacking (`LD_PRELOAD`), Linux Kernel Module (LKM) rootkits, and eBPF rootkits. First, we will explore the userland rootkit technique, `LD_PRELOAD`. We will describe it, provide examples of its usage in the wild, and explain how to detect it.

Linux dynamic linker

Before we delve into the technique itself, let's first understand what the Linux dynamic linker is.

In modern operating systems like Windows and Linux, programs can be linked statically or dynamically. Statically linked binaries are compiled together with all dependencies (libraries) needed for execution. Dynamically linked binaries use shared libraries located on the operating system. These libraries will be resolved, loaded, and linked at runtime. The Linux component that is in charge of this operation is the **dynamic linker**, also known as `ld.so` or `ld-linux.so.*`.

Check it out yourself:

Let's look at the `ls`binary.

1. The `ldd` command allows us to inspect an ELF's dependencies and shared libraries. Open your terminal and run`ldd ls`. In the output, we can see that the `ls`binaryuses `libselinux`, `libc`, and `libpcre` libraries. The first listed dependency is the virtual dynamic shared object, a compact shared library that is automatically mapped into the address space of all user-space applications by the kernel. The last listed dependency is the dynamic linker location.

`ldd ls` output

2. Next, run `strace ls`. In the `strace` output below, the libraries are loaded into memory upon execution.

`strace ls` output

If we take another look at the output, the system checks for the existence of `/etc/ld.so.preload` (five lines above the first red box) prior to `libselinux` being loaded. This leads us to the next section on `LD_PRELOAD`.

LD_PRELOAD

The Linux dynamic linker provides a significant capability called `LD_PRELOAD`. `LD_PRELOAD` holds a list of user-specified, ELF-shared objects. It enables users to load these shared objects into a process's address space before any other shared library and prior to the execution of the program itself. This feature serves multiple purposes, including debugging, testing, and runtime instrumentation, and can be used by writing to the `/etc/ld.so.preload` file or utilizing the `LD_PRELOAD` environment variable.

While it has many legitimate uses, `LD_PRELOAD` can also be leveraged by threat actors as it allows the overwriting of existing functions used by dynamically linked programs. This capability also enables them to evade detection, intercept secrets, and generally alter system behavior.

Check it out yourself:

When examining the `ls` source code, we can see the use of `libc`’s `readdir`function. `ls` reads the directory entries one by one using the `readdir` function inside a loop.

`ls` source code snippet

The `readdir` function returns a pointer to a dirent struct which contains information about the directory entry, such as the name. Once it returns NULL, it indicates the end of the directory.

Let’s create a library that modifies the `readdir` function to hide a file called "malicious_file", compile it, and add it to `LD_PRELOAD`:

1. Create the directory `/tmp/working-dir-test`and copy the code below to the `hijackls.c` file under this directory:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <dirent.h>
#include <string.h>
// Function pointer typedef for the original readdir ls function
typedef struct dirent* (*ls_t)(DIR*);
// Interposed ls function
struct dirent* readdir(DIR* dirp) {
  // Get the original readdir address
  ls_t original_readdir = (ls_t)dlsym(RTLD_NEXT, "readdir");
  struct dirent* entry;
  do {
    // Call the original ls function to get the next directory entry
    entry = original_readdir(dirp);
    // Check if the entry is the file we want to hide
    if (entry != NULL && strcmp(entry->d_name, "malicious_file") == 0) {
      // Skip the file by calling the original ls function again
      entry = original_readdir(dirp);
    }
  } while (entry != NULL && strcmp(entry->d_name, "malicious_file") == 0);
  return entry;
}
```

In the preload library code above, we define a `readdir` function which acts as an interposed function and is called instead of the original `readdir` when the `ls` command is executed. Within our interposed `readdir` function, we obtain the address of the original `readdir` function using `dlsym`, and then call it to get the next directory entry. We compare the name of each entry with "malicious_file" and skip it if it matches, effectively hiding that file from the `ls` output.

`dlsym` allows us to obtain the address of a function within a shared object/library at runtime. Using RTLD_NEXT handle in *dlsym*, we can find and invoke the original `readdir` function.

2. Compile the code to a shared object: `gcc -shared -fPIC -o /tmp/working-dir-test/libhijackls.so /tmp/working-dir-test/hijackls.c -ldl`.

3. Create a directory under `tmp` and fill it with items: `mkdir /tmp/ld-preload-test && cd /tmp/ld-preload-test && for i in file_1 file_2 malicious_file; do touch $i; done;`.

4. Run `ls` and examine all the directory entries.

5. Export `LD_PRELOAD` to the location of the shared object: `export LD_PRELOAD=/tmp/working-dir-test/libhijackls.so`.

6. Run `ls` again. You will see that we successfully hijacked the `readdir` function and the output does not contain “malicious_file”.

7. Finally, unset the environment variable by running `unset LD_PRELOAD`.

Difference between `LD_PRELOAD` and `/etc/ld.so.preload`

`/etc/ld.so.preload` is a system-wide configuration file that applies to all processes and affects the entire system. Access to this file requires root permissions. `LD_PRELOAD`, on the other hand, is an environment variable that allows individual users to specify libraries to be preloaded for a specific executable or command on a per-process basis. Therefore, you don't need to be root to use it. Libraries defined by `LD_PRELOAD` are loaded prior to those in `/etc/ld.so.preload`.

Usage in the wild

The dynamic linker hijacking rootkit technique has been used by numerous threat actors. Whereas some have generated the logic from scratch, others have used publicly available open-source tools. Here are a few examples:

- Winnti for Linux – This Chinese-attributed threat is composed of a user-mode `LD_PRELOAD` rootkit and a main backdoor. The user-mode rootkit is heavily based on the open-source Azazel rootkit. Azazel is used to hide processes, network connections, files, and directories, and also contains backdoor capabilities. Winnti for Linux leveraged Azazel to conceal the main backdoor's malicious activity.
- TeamTNT – This group used libprocesshider in different campaigns. Libprocesshider is an open-source tool designed to hide specific processes from commonly used process-listing tools such as ps, top, and lsof by overwriting the `readdir` function. This technique enabled TeamTNT to conceal XMRig cryptomining and other malicious processes.
- Symbiote – Unlike previous examples where the rootkit was an open-source-based side artifact responsible for hiding other malicious campaign activity, Symbiote is both a backdoor and a rootkit written from scratch. Symbiote conceals its malicious activity by hooking functions in `libc` and `libpcap`. It also uses these capabilities to harvest credentials by hooking `libc`'s `read` function and checking whether the process that is calling it is `ssh` or `scp`. Symbiote leverages its rootkit capabilities for remote access to the machine by hooking Linux Pluggable Authentication Module (PAM) functions and bypassing the authentication mechanism with a hardcoded password match.
- OrBit – OrBit is a dynamic linker hijacking rootkit, consisting of a dropper and a malicious shared object. The dropper is tasked with ensuring the shared object is loaded before any other object. To ensure that, OrBit uses two techniques: adding the object path to `/etc/ld.so.preload` and patching the loader's binary by replacing the string of `/etc/ld.so.preload` to a dedicated path provided by the malware. Similar to Symbiote, OrBit hooks functions in `libc`, `libpcapm`, and PAM to harvest credentials, evade detection, gain persistence, and provide remote access.

Detecting `LD_PRELOAD` abuse

As we saw in the previous examples, attackers use `LD_PRELOAD` to hook different user-land functions and make it hard to investigate infected machines. The following detection methods can help you determine whether you have been infected with this type of rootkit:

- For `/etc/ld.so.preload`: changes in the file will be written to the disk. It is recommended to inspect this file with an image snapshot. If you notice an unusual library path, examine it.
- For `LD_PRELOAD`: search for processes executed with an unexpected `LD_PRELOAD` environment variable (all environment variables per process are located under the `/proc/{pid}/environ file`). If you notice an uncommon library path, examine it.
- Compare the runtime filesystem to the image snapshot. If there's a difference, these files might be part of an attack hidden from certain commands.
- If you are using a runtime detection tool on containers, make sure it supports drift execution libraries loaded into memory. Drift execution detects executable files that have been added or modified after deploying a container.
- Utilize tools like unhide. Unhide uses different brute force techniques to detect concealed processes.

Moreover, the Wiz runtime sensor detects `LD_PRELOAD` attacks. See the following example of an alert detailing a modification to `/etc/ld.so.preload`:

Wiz runtime sensor

Learn more about the Wiz runtime sensor.

Summary

Dynamic linker hijacking via `LD_PRELOAD` is a Linux rootkit technique utilized by different threat actors in the wild. Attackers who successfully deploy this rootkit have powerful control over the infected resource and can benefit from numerous capabilities such as hiding malicious activity and intercepting functions for credential harvesting.

In this blog post, we learned how this rootkit works and provided best practices on how to detect it on your operating system.

Stay tuned for part 2 of this series, where we will delve into Linux Kernel Module (LKM) rootkits.

This blog post was written by Wiz Research, as part of our ongoing mission to analyze threats to the cloud, build mechanisms that prevent and detect them, and fortify cloud security strategies.

Tags

#Research #Security