Rootkits

By Vishay Singh and
Brandon Jaipersaud

Presentation Outline

01 What is a Rootkit? Why Should you Care?

02 Different types of Rootkits

03 Stages of a Rootkit Attack

04 Defense Techniques

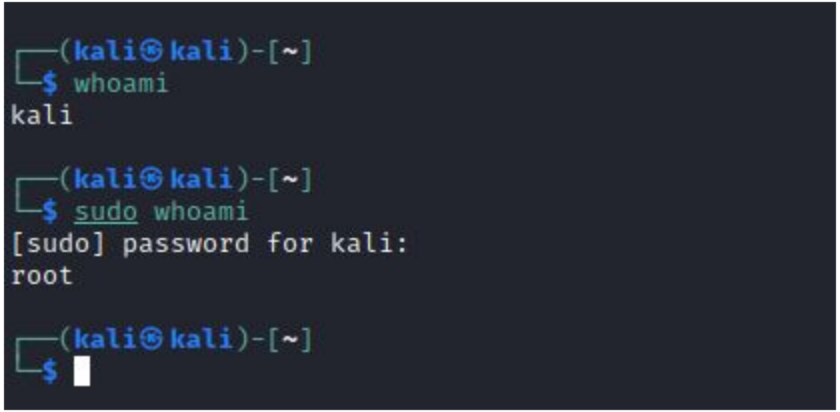
05 LD_PRELOAD Rootkit

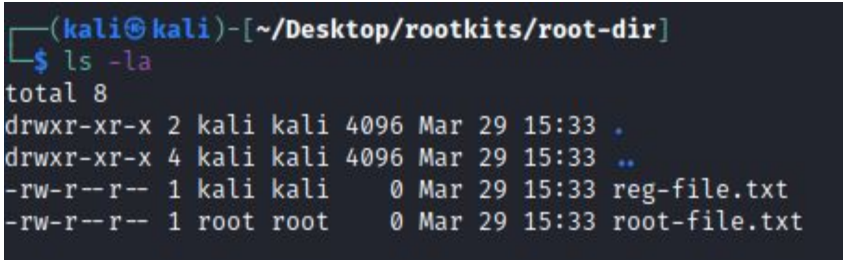
06 Rootkit Attacks in History

What is a Rootkit?

What is a Rootkit?

- Malware designed to give attacker control over the system
- “Root” + “Kit”
- Malicious?
- Staying hidden





Why should you care?

Why should you care?

- Open a backdoor giving the attacker remote access
- Software keylogger
- Surveillance and spying
- Helps other malware infect your system
- These are all long-term effects

Different types of Rootkits

User Level Rootkits

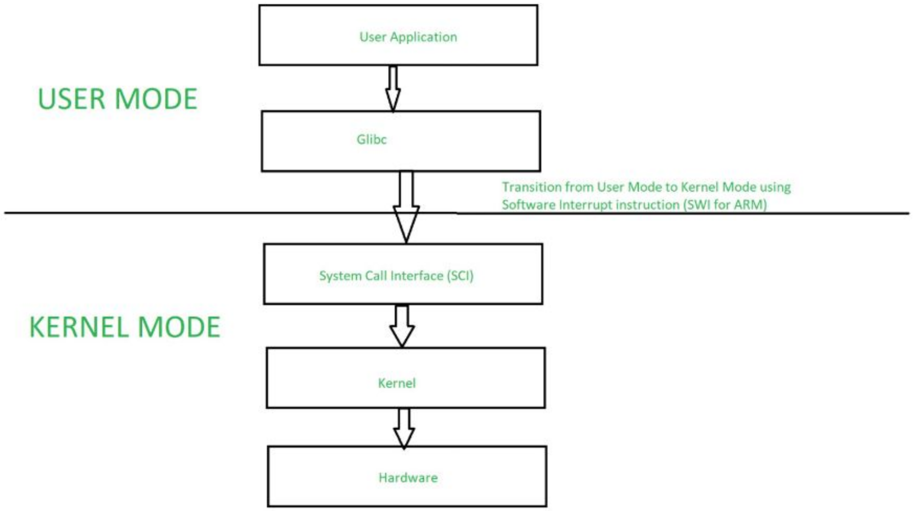
- Think of it like a malicious process running with root privileges
  - UID bit = 0

- Thought Experiment: Imagine you are an attacker that has the ability to inject a program onto a users' system which runs with root privileges. What kind of things would you want this program to do?

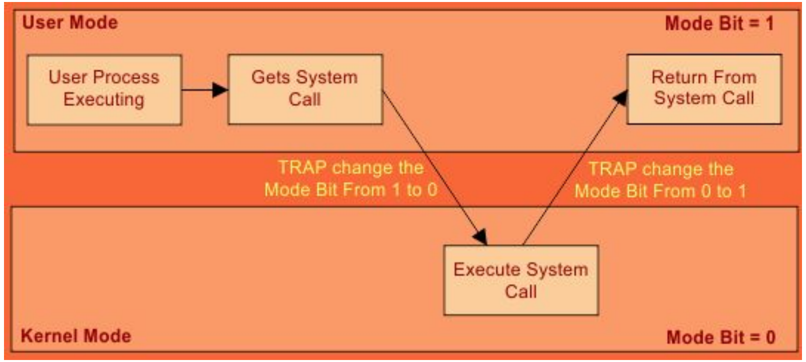
User Level Rootkits

- Process should run every time the system boots up
- Stay hidden
  - Hide from ps, ls -la, lsof, ldd, top and netstat
  - Malicious file should not appear as a hidden file (i.e. .malicious_program.out)
  - Hide activity from logs
- Open a backdoor
  - Modify sshd, login
- Communicate with Command and Control (C&C) server

User Mode vs. Kernel Mode



User mode to Kernel Mode switching



Disadvantages of User Level Processes

- User-mode vs. kernel-mode
- Cannot directly access hardware resources such as file-system, sockets, processor scheduling, memory
  - Need system calls
- Accessing privileged CPU instruction causes trap to kernel/OS.
- Can we do better?

What is more powerful than a user level rootkit?

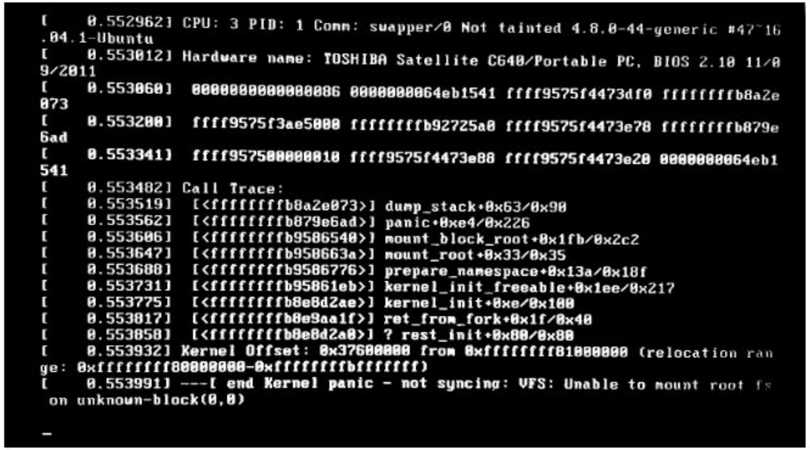
Kernel Level Rootkits

- A rootkit is not just “root” + “kit”
- Runs on same level as operating system
- Attacker has full control over the system
- More difficult to detect and remove
- How does the attacker get to the kernel?
  - Linux Kernel Modules (LKM)

A Disadvantage of Kernel Level Rootkits

- More likely to cause unstable system behavior
  - Kernel panic
  - Blue screen of death

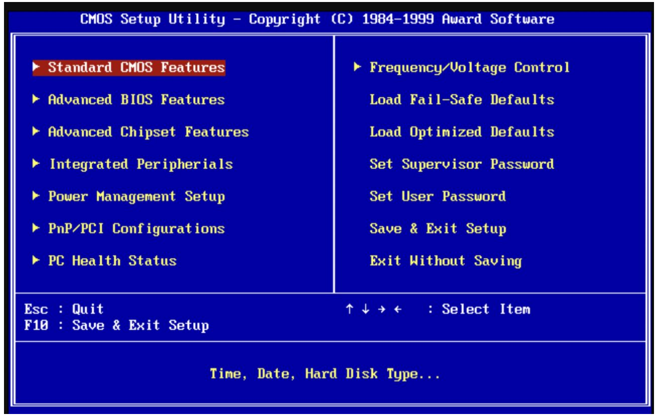
- Think about what happens if a buffer overflow occurs in the kernel



Other Types of Rootkits

Hybrid

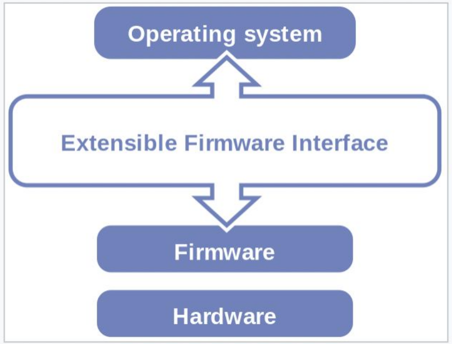
User-Level + Kernel-Level aspects



Firmware

Infect UEFI/BIOS

LoJax - First Discovered UEFI Rootkit in 2018

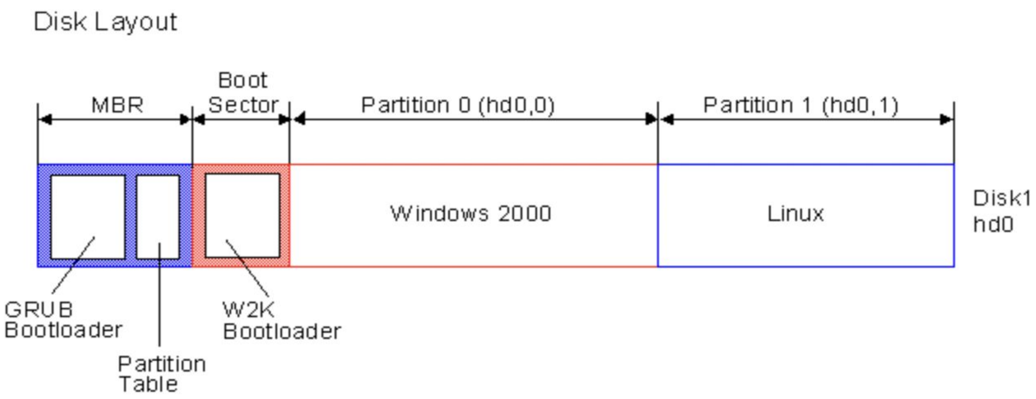


EFI's position in the software stack

Other Types of Rootkits

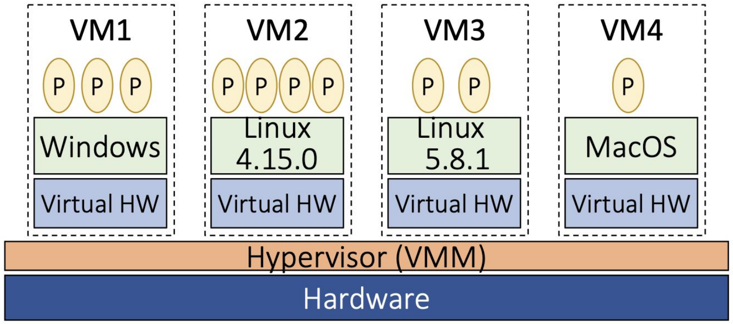
Bootkits

Infect Master Boot Record which helps the computer boot up



Hypervisor/Virtual Machine

Intercept hypervisor emulation of processor instructions



Stages of an Attack

Stages of an Attack

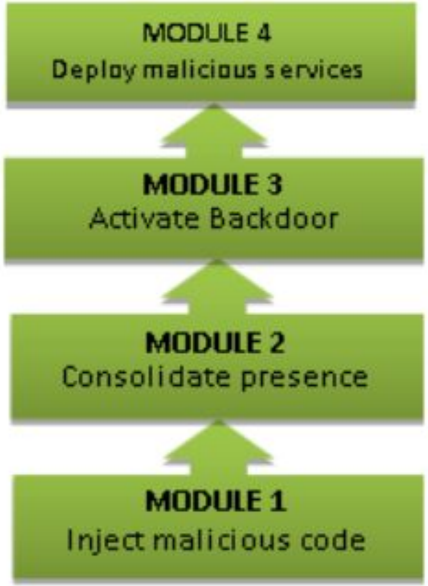


Fig. 2. The Functional Rootkit Stack.

Injection

Before a rootkit can cause havoc on your system, it must first find a way into the system!

Injection

Dropper

Find entry point into system.

Phishing
Social Engineering
Password Cracking

Loader

Exploit vulnerability and inject rootkit with elevated privileges

Buffer overflow
Linux Kernel Modules
LD_PRELOAD ENV Variable

Stealth & Evasion

- OS Dependant
- Permissions (Win.)

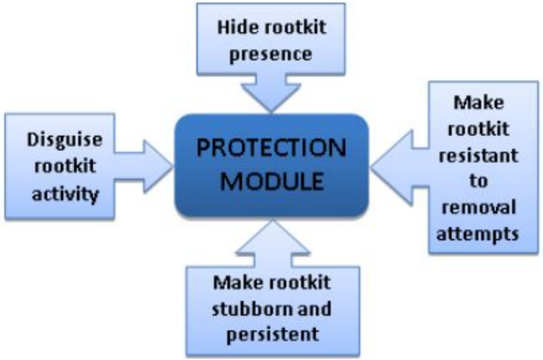


Fig. 3. Various strategies employed at the rootkit protection module.

Stealth & Evasion

- Direct code/data modification (Win.)
- Direct kernel object manipulation (Win.)
- Data structure manipulation (both)
- Hooking (more on this later...)

Stealth & Evasion

- Patching (both)
- Virtualisation (both)
- Evasion vectors

```
Address    Instruction
004937F7   MOV EAX,200
004937FC   MOV EDX,50
00493801   ADD EAX,67F0
00493806   MOV ECX,490AB3
0049380B   JMP 00497000

;Pretend there is a lot of code inbetween here.

00497000   DEC EDX
00497001   MOV DWORD [49E6CC],EDX
00497007   MOV EAX,EDX
```

Jump to address 497000

then continue the code.

Carrying out Malicious Activity

- Backdoor remote access
- Download more malware
- Key logging
- Surveillance
- Distribution of malware

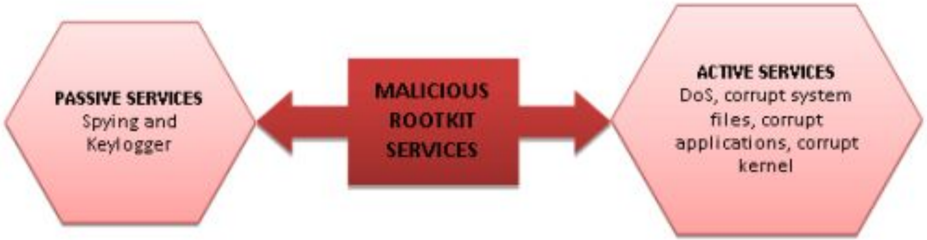
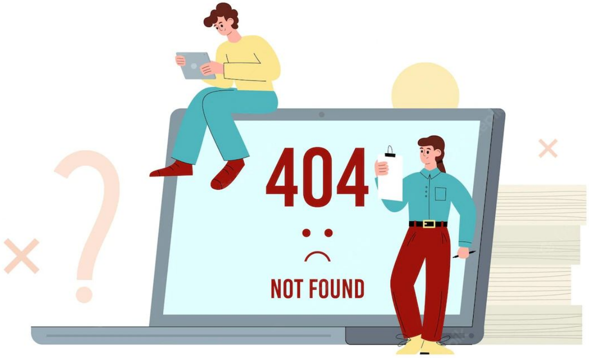


Fig. 5. Malicious rootkit's Active and Passive services.

Defending against rootkits

Detection

- General detection:
  - Unstable behaviour (crashes)
  - Unusual web browser behaviour
  - Performance issues
  - OS settings changes
  - Broken web pages/network activity

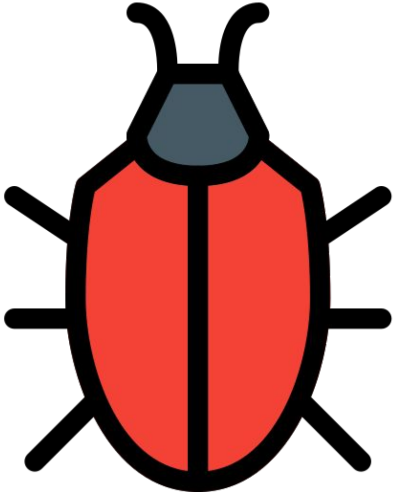


Detection

- User level rootkits:
  - Alternative trusted medium

Detection

- User level rootkits:
  - Alternative trusted medium
  - Virus signatures



Detection

- User level rootkits:
  - Alternative trusted medium
  - Virus signatures
  - Integrity checking

Detection

- User level rootkits:
  - Alternative trusted medium
  - Virus signatures
  - Integrity checking
  - Difference-based detection

Detection

- User level rootkits:
  - Alternative trusted medium
  - Virus signatures
  - Integrity checking
  - Difference-based detection
  - Behavioural detection

Detection

- Kernel level rootkits (much harder):
  - Analyzing the System Call Table
  - Forensic scanning memory dumps

Detection

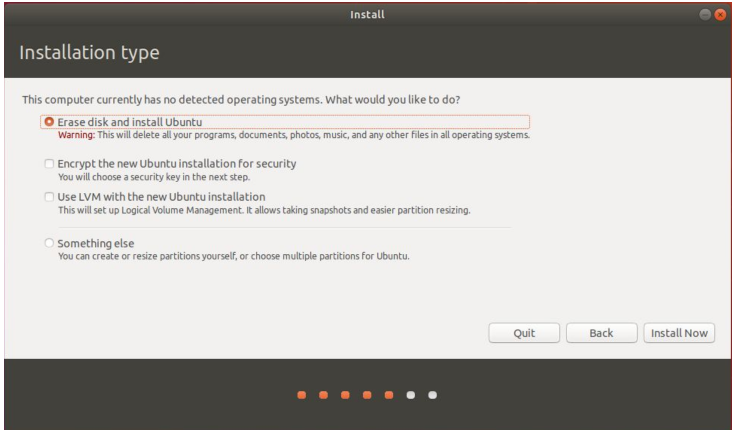
- Kernel level rootkits (much harder):
  - Analyzing the System Call Table
  - Forensic scanning memory dumps
  - Unix tools: Zeppoo, chrootkit, rkhunter, OSSEC
  - Windows tools: RootkitRevealer, F-Secure, Radix, etc.
  - Antivirus software: Avast, Kaspersky, etc.

Limitations

- Arms race between rootkit authors and detectors
  - Detector finds rootkit ⇒ author adapts their code
- Storage analysis may not find evidence of rootkit
  - Inactive rootkit ⇒ suspicious behaviour undetected!
  - Scanners can be ineffective against kernel level rootkits

Mitigation

- Once detected, how can we mitigate and remove the rootkit?
- User level rootkits:
  - Rootkit removal software
  - Boot-time scanning
  - Wipe all drives and reinstall the OS



Mitigation

- Once detected, how can we mitigate and remove the rootkit?
- Kernel level rootkits:
  - All bets are off
  - Wiping drives and reinstalling the OS may not be sufficient!
  - More on this later...
- What does this mean?
  - Prevention is the best defence!

Prevention

- Use scanners
  - Which kind of rootkit are they effective against?
- Avoid phishing attempts
- Update your software!
- Use antivirus software
- Monitor network traffic

Avast One

LD_PRELOAD Rootkit

LD_PRELOAD Rootkit

User-space rootkit based on exploiting LD_PRELOAD ENV. variable

Many user-space rootkits are based on exploiting LD_PRELOAD

🙈 user mode rootkits

- https://github.com/mempodippy/vlany

  Linux LD_PRELOAD rootkit (x86 and x86_64 architectures)

- https://github.com/unix-thrust/beurk

  BEURK is an userland preload rootkit for GNU/Linux, heavily focused around anti-debugging and anti-detection.

- https://github.com/chokepoint/azazel

  Azazel is a userland rootkit based off of the original LD_PRELOAD technique from Jynx rootkit.

- https://github.com/chokepoint/Jynx2

  JynxKit2 is an LD_PRELOAD userland rootkit based on the original JynxKit.

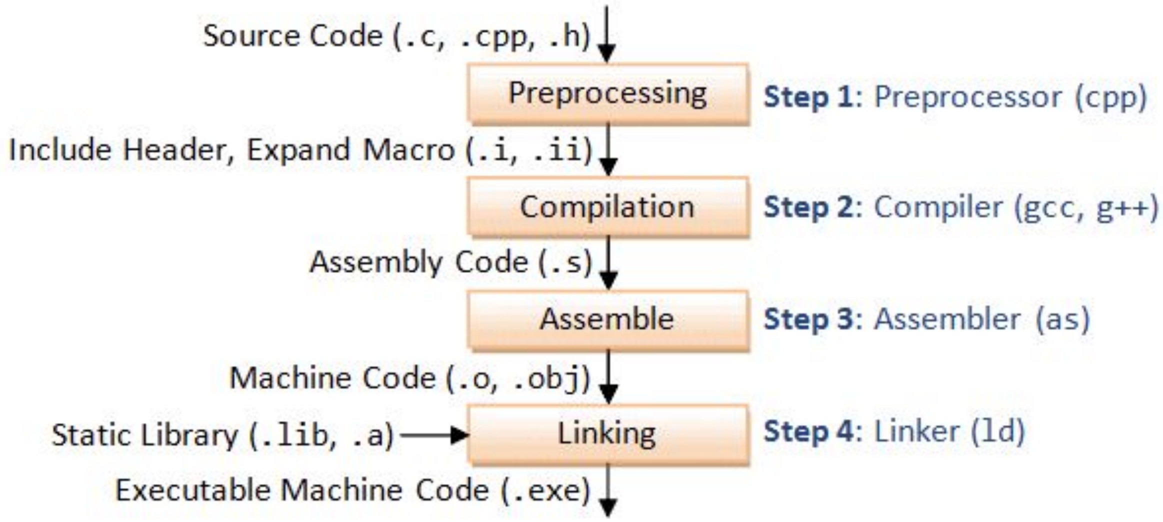
- https://github.com/chokepoint/jynxkit

  JynxKit is an LD_PRELOAD userland rootkit for Linux systems with reverse connection SSL backdoor

- https://github.com/NexusBots/Umbreon-Rootkit

So, How Does it Work?

Compilation Process



Static Linking vs. Dynamic Linking

Static Linking

Libraries linked at compile-time

Located within executable file

1 copy per executable that uses it

Changes to files require re-linking and recompiling

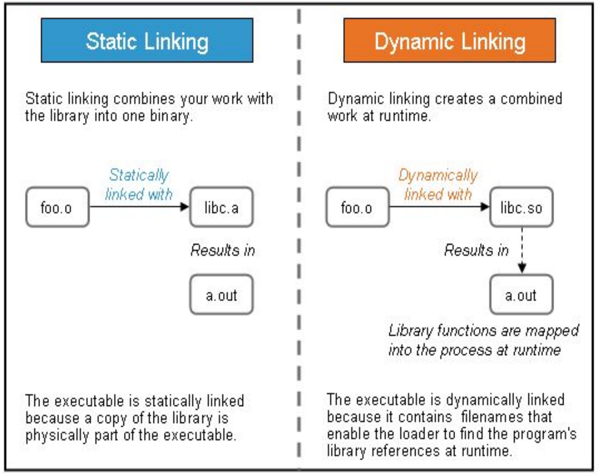
Dynamic Linking

Libraries linked at run-time

Located outside of executable file

Only 1 copy exists system-wide

No re-linking and recompilation needed when file changes are made



Dynamic libraries have a "*.so" naming convention and static libraries have an "*.a".

Dynamic Linker/Loader

LD.SO(8) Linux Programmer's Manual LD.SO(8)

NAME top

ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS top

The dynamic linker can be run either indirectly by running some dynamically linked program or shared object (in which case no command-line options to the dynamic linker can be passed and, in the ELF case, the dynamic linker which is stored in the **.interp** section of the program is executed) or directly by running:

*/lib/ld-linux.so.** [OPTIONS] [PROGRAM [ARGUMENTS]]

LDD(1) Linux Programmer's Manual LDD(1)

NAME top

ldd - print shared object dependencies

SYNOPSIS top

ldd [*option*]... *file*...

DESCRIPTION top

ldd prints the shared objects (shared libraries) required by each program or shared object specified on the command line. An example of its use and output is the following:

LDD

Uses the dynamic linker (ld.so) to print shared libraries used by an executable

```
brandon@ubuntu20-04:~$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffd18dd3000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fad90b90000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fad90998000)
        libpcre2-8.so.0 => /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007fad90908000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fad90900000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fad90c08000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fad908d8000)
```

See Anything Familiar?

libc

- C standard library is dynamically linked at runtime
- This is why when we compile and run C programs we don't need to explicitly link the C std library

```c
#include <stdio.h>

int main(int argc, char **argv)
{

    printf("Hello world!")

}
```

Dynamic Linking and LD_PRELOAD

We can use LD_PRELOAD to dynamically load our custom library into every executable that is run!

```
┌──(kali㉿kali)-[~/Desktop/rootkits]
└─$ gcc -g -fPIC  *.c -shared -o my-lib.so
```

```
┌──(kali㉿kali)-[~/Desktop/rootkits]
└─$ export LD_PRELOAD=$PWD/my-lib.so
```

```
┌──(kali㉿kali)-[~/Desktop/rootkits]
└─$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffd143f4000)
        /home/kali/Desktop/rootkits/my-lib.so (0x00007f28b29b8000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f28b2970000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f28b27a0000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f28b2708000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f28b2700000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f28b29e8000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f28b26d8000)
```

Dynamic Linking and LD_PRELOAD

```
brandon@ubuntu20-04:~$ strace /bin/ls
execve("/bin/ls", ["/bin/ls"], 0x7ffe58050b40 /* 59 vars */) = 0
brk(NULL)                               = 0x560a5c8cd000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe31dc2cb0) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=98806, ...}) = 0
mmap(NULL, 98806, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6fde428000
close(3)                                = 0
```

Notice:

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

```
/etc/ld.so.preload
       File containing a whitespace-separated list of ELF shared
       objects to be loaded before the program.  See the
       discussion of LD_PRELOAD above.  If both LD_PRELOAD and
       /etc/ld.so.preload are employed, the libraries specified
       by LD_PRELOAD are preloaded first.  /etc/ld.so.preload has
       a system-wide effect, causing the specified libraries to
       be preloaded for all programs that are executed on the
       system.  (This is usually undesirable, and is typically
       employed only as an emergency remedy, for example, as a
       temporary workaround to a library misconfiguration issue.)
```

How can we exploit LD_PRELOAD?

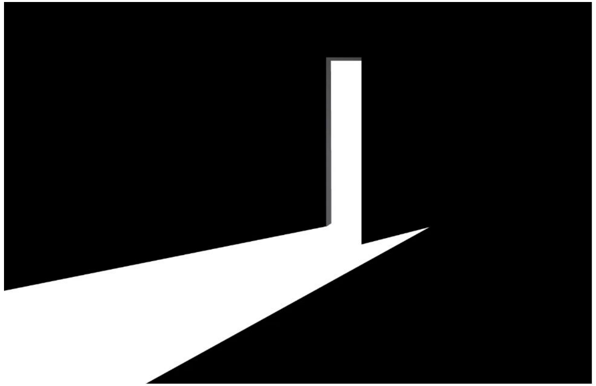
Exploiting LD_PRELOAD

We now have a way to inject a malicious dynamic (shared) library into every running process. Every program that is executed will have our library injected into it!

We can use this to do things like:

- Open a backdoor that we can remotely access
- Hide our malicious files and network connections from the output of commands like netstat and ls

How?



Function Call Hooking

We can replace functions from the C standard library with our own

Our hooked function call should:

- Behave normally to other users
- Execute malicious functionality when called by attacker
- Will see more in the demo!

```c
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>

int puts(const char *message)
{
    int (*new_puts)(const char *message);

    int result;
    new_puts = dlsym(RTLD_NEXT, "puts");
    if(strcmp(message, "Hello world!\n") == 0)
    {
        result = new_puts("Goodbye, cruel world!\n");
    }

    else
    {
        result = new_puts(message);
    }

    return result;
}

```

System Call Hooking

Used in kernel-level rootkits

Hooks system calls rather than function calls
Ex. malloc() vs. brk()

Inject into the kernel via. Linux Kernel Module

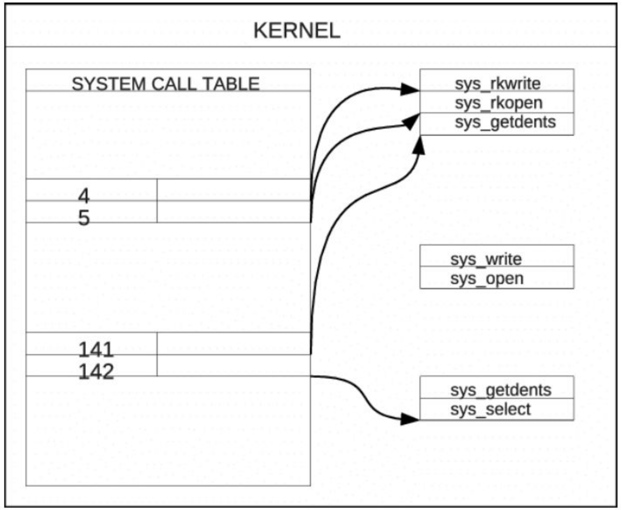
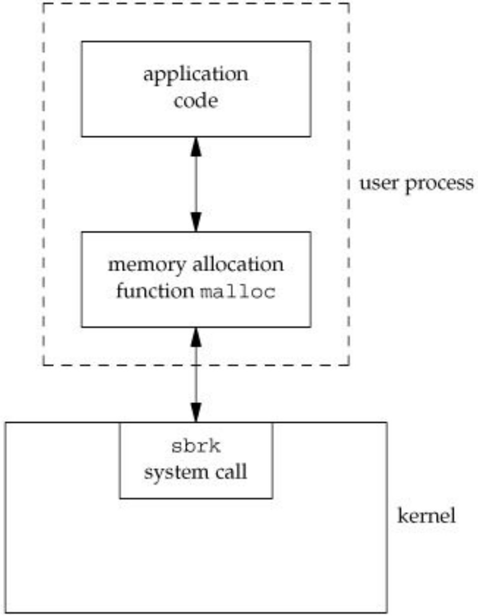


Figure 1.11. Separation of malloc function and sbrk system call



Demo

Demo

You now have the basic ideas behind LD_PRELOAD rootkits.

You will now see a rootkit attack in action! It will show things like:

- How write() can be hooked to open a remote backdoor
- Hiding from output of ls and netstat

Examples from history

A General Timeline

- 1990 and 1999 first rootkits made for Unix-based and Windows OS respectively
- 2004 Greek Watergate rootkit
- 2009 Machiavelli rootkit
- 2010 and 2012 state funded rootkits
- Modern day rootkits continue to advance

https://www.avast.com/c-rootkit

Sony DRM rootkit (2005)

- 22 million CDS affected
- DRM (anti-piracy software), which installs itself
- The DRM modified the OS to stop CD copying
- The DRM could not be easily uninstalled, and hid its existence
- Sony did not mention the software in the EULA (user agreements)

https://www.csoonline.com/article/2998952/sony-bmg-rootkit-scandal-10-years-later.html

LoJax rootkit (2018)

- First rootkit to run within the UEFI (BIOS)
- Can execute malicious code on disk during boot process
- Circumvents OS reinstall AND hard drive replacement!
- Removal involves flashing UEFI firmware
- Advanced persistent threat (APT) group, Fancy Bear

https://www.crowdstrike.com/blog/who-is-fancy-bear/

Scranos rootkit (2019)

- Rootkit that steals passwords and payment information stored in the victim's browser
- Creates a botnet of the infected devices
- Used for liking/subscribing on YouTube, and distribution of third party malware

https://www.bitdefender.com/blog/labs/inside-scranos-a-cross-platform-rootkit-enabled-spyware-operation/

Thanks for listening

Created by
Brandon Jaipersaud
and
Vishay Singh

References

- https://www.avast.com/c-rootkit#:~:text=A%20rootkit%20is%20a%20malicious,manipulate%20it%2C%20and%20steal%20data
- https://www.ijais.org/research/volume10/number8/anande-2016-ijais-451540.pdf
- https://scholar.afit.edu/cgi/viewcontent.cgi?article=4107&context=etd
- https://www.cs.ru.nl/bachelors-theses/2020/Egidius_Mysliwietz___1000796___Identifying_rootkit_stealth_strategies.pdf
- https://www.kaspersky.com/resource-center/definitions/what-is-rootkit
- https://www.youtube.com/watch?v=0LvF0KtBWxY
- http://www.mhprofessional.com/downloads/products/0071591184/0071591184_chap10.pdf
- https://blog.malwarebytes.com/how-tos-2/2020/01/how-to-prevent-a-rootkit-attack/#:~:text=To%20fully%20protect%20yourself%20against,then%20reinstall%20the%20entire%20system.&text=Phishing%20is%20a%20type%20of,or%20downloading%20an%20infected%20attachment
- https://en.wikipedia.org/wiki/Rootkit
- https://en.wikipedia.org/wiki/Sony_BMG_copy_protection_rootkit_scandal
- https://medium.com/swlh/linux-basics-static-libraries-vs-dynamic-libraries-a7bcf8157779#:~:text=Dynamic%20or%20shared%20libraries%20occur,library%27s%20files%20at%20compile%20time
- https://www.zdnet.com/article/fancy-bear-lojax-campaign-reveals-first-documented-use-of-uefi-rootkit-in-the-wild/
- https://techcrunch.com/2019/04/16/scranos-rootkit-passwords-payments/?guce_referrer=aHR0cHM6Ly93d3cuYXZhc3QuY29tLw&guce_referrer_sig=AQAAAGV8hFHt0511UTKkFSEGkn2qfocQdm0oPtdOOU8aCAlmSt_xxifxGbko-wsBpwKFzQ-6bZxmwf0ew3g7Fm-CqHhfCdTvSHFa4eLS19CZ2plBB9D5mG9xCsy5oYr-JUc-cQhqu2NaAjpmYkrn604tHo59Id_P3L7UI_JfBRqyX1sG