Fall
2016

Technical Report

ADOBE PRACTICUM TEAM - RAHUL GAJRIA, APURV BHARGAVA, UDIT GUPTA, SHIVANI SINGHAL

ENCLOSED IN THIS DOCUMENT IS THE TECHNICAL REPORT OF THE DOCKER CONTAINER SECURITY PROJECT SPONSORED BY ADOBE

Contents

Introduction

Virtual machines(VMs) have been around for half a century and are extensively used to sandbox various applications running on a host. VMs simulate an entire Operating System on top of a host.

Container technology, being a relatively new one, was introduced in Linux in 2008. The agenda for this technology was to provide a sandboxing environment to run applications similar to virtual machines without the overhead of running a hypervisor over which each machine would run its own kernel. Containers would run as separate processes on the same host, yet simulate the virtual environment on a shared OS and file system.

The major difference between a VM and a container comes from the isolation level. VMs provide isolation through hardware level abstraction, they emulate an OS completely by having their own BIOS, to CPU, to disk storage, a complete OS while containers share the same Operating system and provide process and user space level isolation.

The motivation behind using containers may vary depending on the use case. It may consist of decreasing power/storage costs, easing the lives of developers to test and ship code or just sandboxing individual applications deployed on VMs. With the growing usage of containers, a major issue still lies in the fact that not a lot of security effort has been put into developing security best practices for container usage. Due to this shortcoming, container technology is yet to be pervaded across production environments.

To run a container securely, one needs to go through numerous steps. A container has many attack surfaces, which if not correctly configured, can be vulnerable to attackers -

- The host running the containers.
- The shared libraries of the host that are used by the containers.
- Docker daemon.
- Network connections to the container.
- Docker images themselves

An out-of-the box installation of just about any Linux distribution would be capable of running the Docker daemon and Docker containers but it would leave your host exposed to many security and performance concerns.
There are three main steps that are required to configure a basic host for production -

- Strip the operating system of any extra services and software so that only the tools and services required to run Docker are up and running.

- Install and configure the Docker daemon to run your containers. This includes both performance and security settings that configure Docker to be more suitable for a production environment rather than a development environment.
- Configure the firewall to only allow incoming traffic for SSH by default and open ports on-demand that are required by the containers for external communication.

Motivation

Container technology is around eight years old and docker came into picture only three years back, which implies it has only been adopted recently and hence robust security measures are yet to be taken to provide proper isolation and sandboxing of docker containers. Container technology seems promising as it would make the applications really light weight and make deployment into production environment quite simple.

For the same reason companies are moving towards widespread adoption of container technology to get their production environments up and running while supporting multi-tenancy.

Since multiple containers would run as processes directly on the same host, the attack surface becomes very large. One single malicious container can compromise the host and the other containers running on it. It may become the cause of denial of service attacks or information leakage.

Even though containers have been around from a long time, the primary hindrance in widespread adoption was the difficulty in its deployment. Docker abstracts a lot of these administrative tasks and provides a set of easy to use APIs, leading to decrease in learning curve and significant increase in adoption. These two reasons were our driving factors for choosing to analyze the security posture of it. Additionally, Docker provides a default set of measures for protection of host system from malicious containers like cgroups, namespaces, capabilities and seccomp profiles.

We are going to build up on these basic templates to further analyse Linux system calls for seccomp profiles, capabilities.

Related work

Although containers have become popular in the last couple of years due to ease of shipping and interoperability, the security of containers is still a major hindrance to enterprises which

prevents them from deploying this technology in the production environment. As a result, docker community has been actively involved in identifying security loopholes in docker containers and number of startups have been floating around to look into container security like twistlock and aqua.

In [15] authors talk about four major areas to consider when reviewing docker security - intrinsic security of the kernel and its support for namespaces and cgroups, the attack surface of the docker daemon itself, loophole in the container configuration profile, either by default, or when customized by users and the "hardening" security features of the kernel and how they interact with containers.

In [16], authors have talked about vulnerability exploitation in docker containers and have provided a comparison with exploitation of application vulnerabilities in virtual machines. Significant amount of work has been done in the domains of SELinux [17], GRSecurity [18], PaX [19] and Capabilities [20] on which we have capitalised and came up with our own analysis. Moreover, significant strides have been made in identifying loopholes in ASLR implementation and effective mitigation strategies have been provided for the same as mentioned in [11]. Finally, the NCC document [6] provide a brief overview about understanding and hardening linux containers which helped us in getting started with this project.

Analysis

Our analysis for docker container security is divided into five parts. We first analyze the Linux system calls to create custom seccomp profiles based on diverse functionality. Next we analyze the Linux Capabilities whitelisted by docker. We go on to analyze how can GrSecurity and Pax can help in system hardening and lastly we analyze the role selinux plays in security of Linux systems. We further present these analyses in detail as follows:

Seccomp

There are a large number of system calls that are exposed for use by every user land process. Majority of these go unused for even the entire lifetime of the process. The absence of use however, does not guarantee absence from exploitation. As system calls evolve and mature, bugs are found and eradicated. Some such bugs may also be used to develop 0-day exploits to aid hackers. The huge number of available system calls may provide a large surface to be leveraged by an attacker. In such a scenario, systems can benefit by restricting the available system calls to only bare minimum requirements.

Seccomp profiles provide a means for a process to specify a filter for incoming system calls. This filter is expressed as a Berkeley Packet Filter (BPF), similar to that on socket filters, except that the data operated upon is related to the system call being made: system call number and the system call arguments. The seccomp filter mode allows developers to write BPF programs that determine whether a given system call will be allowed or not.

On the lower level filters can be installed using either seccomp() or prctl(). The BPF program must be constructed first, then installed in the kernel; after that, every system call triggers the filter code. Also, filters cannot be removed once they have been installed, since installing a filter is effectively a declaration that any subsequently executed code is not trusted.

Seccomp in Docker

This feature is available in Docker only if Docker has been built with seccomp and the kernel is configured with CONFIG_SECCOMP enabled. The following command can be used to check if Seccomp is enabled on a container or not.

```
$ cat /boot/config-`uname -r` | grep CONFIG_SECCOMP=
CONFIG_SECCOMP=y
```

Note: seccomp profiles require seccomp 2.2.1 and are only available starting with Debian 9 "Stretch", Ubuntu 15.10 "Wily", Fedora 22, CentOS 7 and Oracle Linux 7. To use this feature on Ubuntu 14.04, Debian Wheezy, or Debian Jessie, you must download the latest static Docker Linux binary. This feature is currently not available on other distributions.

Format

The seccomp profiles in Docker can be implemented as simple json files specifying specific calls to be allowed/ blocked as shown -

```
{
        "defaultAction": "SCMP_ACT_ERRNO",
        "syscalls": [
                {
                        "name": "accept",
                        "action": "SCMP_ACT_ALLOW",
                        "args": []
                },
                {
                        "name": "accept4",
                        "action": "SCMP_ACT_ALLOW",
                        "args": []
                },
                ...
        ]
}
```

Seccomp profiles can be both blacklist driven, or whitelist driven. To implement a blacklist approach, we can set the defaultAction attribute to SCMP_ACT_ERRNO and allow the required syscalls according to need as shown above. Whitelisting would require setting the defaultAction attribute to SCMP_ACT_ALLOW and then denying individual syscalls setting them as SCMP_ACT_ERRNO. It is highly recommended to follow a whitelisting approach. This would make sure that some unaccounted syscalls do not slip through our filters.

Approach

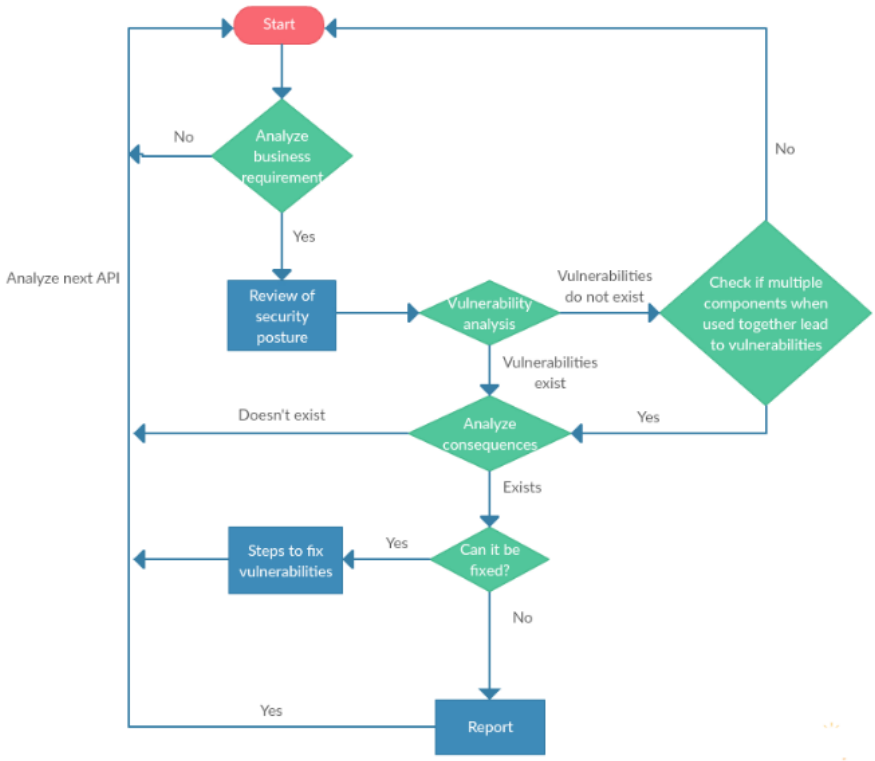
The default seccomp profile disables 44 system calls out of 300+. These can be divided into the following 6 categories -

1. Process Control
2. File Manipulation
3. Device Manipulation
4. Communication
5. Information maintenance
6. Protection

Another point to note here is that just because the rest of the 250+ syscalls are allowed doesn't mean that all of them -

1. Are going to be required in every context.
2. Are free of any bugs/ security vulnerabilities.

To this end we analysed all remaining 250+ syscalls for existing vulnerabilities. The objective of this analysis was to be able to come up with seccomp profiles that could satisfy various business use cases. We followed the flow diagram given below to analyse each one.



The results of this analysis can be found here. We divided the vulnerabilities found into 3 major categories as explained below -

1. **Privilege Escalation** - This category of exploits is used by an attacker to increase the privileges that are assigned to them and execute operations well beyond their allowed scope of actions. Following syscalls are found to be prone to this kind of vulnerability:
    - socket() : This system call creates an endpoint of communication and returns a descriptor. The domain argument specifies a communication domain; this selects the protocol family which will be used for communication.
        - Vulnerability: For Linux versions 4.2 and below, attacker was able to gain elevated access by exploiting a bug.
        - Exploit: If an operation on a particular socket is unimplemented, they are expected to point the associated function pointer to predefined stubs, for example if the "accept" operation is undefined it would point to sock_no_accept(). However, we have found that this is not always the case and some of these pointers are left uninitialized. This issue is easily exploitable for local privilege escalation. In order to exploit this, an attacker would create a mapping at address zero containing code to be

executed with privileges of the kernel, and then trigger a vulnerable operation using a sequence

- Ptrace() : This system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.
    - Vulnerability: A local privileged namespace user can obtain elevated privileges outside of the original namespace on the target system. This vulnerability existed prior to linux kernel versions 4.4.1.
    - Exploit: It could occur when a root owned process tries to enter a user namespace, wherein a user attempts to attach the entering process via ptrace(1). A privileged user could use this flaw to potentially escalate their privileges on the system.

2. **Denial of Service(DOS):** This exploit disrupts the availability of the system in various ways like by using up the memory of the system or causing unavailability of resources. Following two syscalls are examples found to be vulnerable to DoS but were patched later:

    - pipe : The pipe() syscall is used to create a communication channel between two processes. Each pipe has two ends, a reading and a writing end. When the process on the writing end writes the data into the pipe, the linux kernel buffers it until the reading end reads it
        - Vulnerability: Before Linux kernel 4.5, the pipe.c file did not limit the size of unread data in a pipe
        - Exploit: A local user can create many pipes with non default sizes which would cause memory consumption leading to a denial of service. Details for this exploit are available in [3]
    - utimensat: This function changes the last access and last modified timestamps of a file with the precision of nanoseconds
        - Vulnerability: UTIME_NOW is a flag that implies that the file should be given the current timestamp and UTIME_OMIT is a flag that implies that the timestamp of the file should be left unchanged. Linux kernel 2.6.22 and other versions before 2.6.25.3 does not check file permissions when certain UTIME_NOW and UTIME_OMIT combinations are used
        - Exploit: Since there are no permissions checked, local users can modify access times of arbitrary files leading to denial of service to resources which may read data from a file if there is a change in last modified time stamp etc. This vulnerability is documented in CVE database as [4]

3. **Information leakage**: This exploit uses the vulnerabilities in processes to disclose information to unauthorized users or process. Following two syscalls are examples found to be vulnerable to Information leakage but were patched later:
    - accept: The accept() system call is used accept a connection on a socket in Linux
        - Vulnerability:FreeBSD versions from 4.0 until before 4.6 have a few system calls, accept being one of them, vulnerable to signed integer buffer overflow.
        - Exploit: The input accepted is a positive integer, hence if the attacker sends a negative integer, the boundary checking code would fail and may lead to leakage of information from stack of the process that may not be intended for the user. Details for these attack are available at[13]
    - lseek: lseek syscall is used to reposition a read/write file offset of a file descriptor
        - Vulnerability: The kernfs_xread function in kernfs in NetBSD 1.6 through 2.1, and OpenBSD 3.8, does not properly validate file offsets against negative 32-bit values that occur as a result of truncation[14]
        - Exploit: This allows local users to read arbitrary kernel memory and gain privileges via the lseek system call

    There are vulnerabilities that may facilitate the attacker to perform more than one exploits, privilege escalation by default does information leakage.

Findings

Seccomp profiles depend on the syscalls that are going to be used by a system. Since, all programs do not do the same job, they do not require the same syscalls. Consequently, all systems cannot use the same seccomp profile. We, thus, came up with some use cases and developed profiles based on the applications that were going to be run on the container. We used the categorization of syscalls as explained in the previous section to guide our approach in identifying syscalls related to a particular functionality.

For example - Syscalls under the ‘Communication’ category can be removed from a container that requires complete isolation. We created 5 seccomp profiles - 2 based on varying amounts of strictness (extremely lenient to extremely strict) and 3 based on functionality -

1. Network isolation
2. File management
3. Information leakage (System information)

We also identified some syscalls which were mandatory for the working of a container even if they are not used within the container [12](https://github.com/docker/docker/issues/22252). If

these are not allowed, the container will fail to run with varying error messages depending on the missing syscall. These syscalls include -

1. capget
2. capset
3. chdir
4. fchown
5. futex
6. getdents64
7. getpid
8. getppid
9. lstat
10. openat
11. prctl
12. setgid
13. setgroups
14. setuid
15. stat

The seccomp profiles for all five use cases have been uploaded on our github page.

Capabilities

Linux processes are divided into two categories *privileged* and *unprivileged*. The privileged processes run with root permissions, which implies they have complete control over the system. If there are vulnerabilities in such processes they may be leveraged by the attacker to gain control over the entire system.

For example /bin/ping needs to access raw socket which is a privileged operation hence needs to run as root. If there was a vulnerability in /bin/ping program the attacker can leverage it to gain root privileges and own the system.

Many processes do not need all the privileges root owns, hence Linux kernel 2.2 introduced *Capabilities*. Capabilities give fine grained access control to processes. The root privileges were broken down into independent units so that processes could be given only those privileges that were needed by them.

So considering the example of /bin/ping, in this case we would only require to give the process the CAP_NET_RAW capability. In this case even if the attacker exploits the process, (s)he would only be able to get privilege to access raw sockets hence the attack surface is reduced.

That said, *Capabilities* are a contested topic in the Linux community and certain distributions do not find them secure enough to be used. Capabilities add a layer of complexity on the system since if used incorrectly they may give users the false assurance of security.
Docker blocks most capabilities and only enables 14 of them by default. We analyzed these 14 capabilities and found that any bugs found in them were already fixed. Some of these capabilities are dangerous hence processes having them may lead to severe damage if exploited.

Following are some dangerous capabilities and should be refrained from use so far as possible:
CAP_CHOWN: Make arbitrary changes to file UIDs and GIDs
Exploit: If an attacker is able to exploit a file that has this capability (s)he can own system files like /etc/passwd and read all passwords or change root to have no password
CAP_DAC_OVERRIDE: Bypass file read, write, and execute permission checks.(DAC is an abbreviation of "discretionary access control".) Used mainly for mount, umount operations.
Exploit: Because CAP_DAC_OVERRIDE permits to read, write and execute all system files, it is similarly dangerous as described in the previous part.
CAP_FOWNER:

- Bypass permission checks on operations that normally require the filesystem UID of the process to match the UID of the file (e.g., chmod(2), utime(2)), excluding those operations covered by CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH
- set extended file attributes (see chattr(1)) on arbitrary files
- set Access Control Lists (ACLs) on arbitrary files
- ignore directory sticky bit on file deletion
- specify O_NOATIME for arbitrary files in open(2) and fcntl(2)

Exploit: This capability lets a user perform functions on a file that only a file owner may be allowed like chmod. If the attacker is able to exploit the file containing this capability they may change permissions on system files and gain access to data

We further divide the capabilities based on their categories and suggest that users strictly do not use the capabilities which are not required by their system, follow the principle of least privileges.

The categories are:

- **Network communication**
  - CAP_NET_RAW
  - CAP_NET_BIND_SERVICE
- **File system**
  - CAP_CHOWN
  - CAP_DAC_OVERRIDE
  - CAP_FSETID
  - CAP_FOWNER
  - CAP_MKNOD
  - CAP_AUDIT_WRITE
- **Process control**
  - CAP_SETGID
  - CAP_SETUID
  - CAP_SETFCAP
  - CAP_SETPCAP
  - CAP_SYS_CHROOT
  - CAP_KILL

In order to drop a capability docker provides --cap_drop option when spinning a new container For instance if we want to block the CAP_KILL capability while spinning a container we can use the following command
Docker run --cap_drop=KILL [IMAGENAME]

GR SECURITY

GRSec is a mandatory aspect and an extensive security enhancement to the Linux kernel that defends against a wide range of security threats. Before we go on to further describe GRSec in detail, it is important to look into certain concepts like role based access system (RBAC) and significance of policy configuration file.

Role Based Access Control System (RBAC)

Role Based Access Control (RBAC) is an alternative method of controlling user access to file system. It differs from discretionary access control (DAC) or Mandatory Access Control (MAC) in the sense that instead of access being controlled by user permissions, the system administrator establishes 'Roles' based on business functional requirements or similar criteria. Each role defines what operations can be performed on certain objects (as defined under 'Policy Configuration File' heading). These roles have different types and levels of access to objects.

In contrast to DAC or MAC systems, where users have access to objects based on their own and the object's permissions, users in the RBAC system must be members of the appropriate group or role before they can interact with certain files, directories or devices. From an administrative point of view, this makes it easier to control who has access to various parts of the file system, just by controlling their group memberships [1].

Policy Configuration File

RBAC system is managed through a policy file in which we specify a set of rules for hardening of linux systems or by extension docker containers. These policy configuration files are customizable to a great extent where we can define users, roles corresponding to individual users, etc. Below we have listed down important aspects of policy configuration file:

- **Roles:** It is an abstraction that encompasses traditional users and groups that exist in linux distributions. Roles can be used to split the responsibility of system administration into smaller logical sets of responsibilities, such as "database administrator" or "DNS administrator".
- **Objects:** It is a part of the system that is used by the program running on the system. It can be an absolute path to a file or a directory; a capability; a system resource; a PaX flag; network access (IP ACLs)
- **Subjects:** It is a set of directories for which rules need to be added. A subject uses and accesses objects and the ruleset of the subject enforces what objects it may use and in what way.

GRSec is based on role based access system (RBAC) in which we restrict system access to authorized users. We need an RBAC system if we want to restrict access to files, capabilities, resources, or sockets to *all* users, including root. Given that even root doesn't have all the privileges, so even if the attackers are able to gain access as root, the damage can be minimized. Given that as part of our seccomp profiles, one of the primary concern we faced while analyzing system calls was privilege level escalation which resulted in attacker gaining root access, RBAC system would ensure that damage will be minimum since the attack surface for 'root' has been reduced drastically. This is in accordance with principle of least privilege in which we give minimum set of privileges to users to perform the desired task.

RBAC system is managed through a policy file in which we can specify a set of rules for hardening of containers. '**Gradm**' is a tool which allows you to administer and maintain a policy for your system. It is highly customizable and using it you can enable or disable RBAC system, reload the RBAC roles, change an individual user role, etc. The default policy file is installed in the following path: /etc/grsec/policy. This has been described in more details in [2].

Below is a sample policy config file which can be used to manage role based access control:

```
role user1 u
subject /
        /         r
        /opt      rx
        /home  rwx
        /mnt      rw
        …
subject /bin/ping
        +CAP_NET_RAW
        -CAP_NET_BIND
        ...

        -PAX_SEGMEXEC
        -PAX_MPROTECT
        ...

subject    /usr/bin/ssh
        connect 192.168.1.0/24:22 stream tcp

RES_FSIZE 3k 3k
```

Description of above policy config file:

In the above configuration file, a role has been defined for user1 and corresponding to this user, several subjects are defined. Subjects are basically a bunch of files or directories as described in the definition above, and multiple subjects can be defined for each user. In first case subject is defined as directory '/', and a bunch of permissions have been defined for sub-directories inside this subject. In 2nd case subject is defined as a path for 'ping' command, we can add/drop capabilities or PaX flags depending on the requirements. In the 3rd instance, we have defined subject to be the path of 'ssh' command and then it has been restricted to connect only to the given IP over a tcp session. We can also define soft and hard limit for a particular system resource. For instance in the above example, we have defined 3k as the soft and hard limit for file size which indicates that any process during its execution would not be able to create files of size exceeding 3KB.

Advantages of GRSec in hardening containers:

- **Privilege level escalation:** Given that privilege level escalation which resulted in attacker obtaining root access was our primary concern while analyzing system calls for seccomp profiles, RBAC would ensure that even as root, access will be restricted to files, capabilities, processes, etc.
- **DoS attacks:** Given that DoS attacks were another prominent category of attacks we found as part of system call analysis, it would be beneficial to put some sort of limit on

the system resources. Given that policy config file allows us to define hard and soft limit for various aspects, it would not be easy for attacker to consume system resources. Even as 'root', it would be impossible for attacker to overcome hard limit defined in policy config file.

PaX

PaX is a patch for the Linux kernel that implements least privilege protections for memory pages. The least-privilege approach allows computer programs to do only what they have to do in order to be able to execute properly.

PaX adds intrusion prevention mechanisms to the kernel that reduce the risks posed by exploitable memory corruption bugs.

PaX flags data memory as non-executable, program memory as non-writable and randomly arranges the program memory. The former prevents direct code execution absolutely, while the latter makes so-called return-to-libc (ret2libc) attacks difficult to exploit. Hence it claims to effectively prevent many security exploits, such as some kinds of buffer overflows.

Some of the features of PaX are:

- Executable space Randomization
- Address Space Layout Randomization
- Binary Marking

- **Executable space Randomization:**
  - PAX_NOEXEC : **Enforce non-executable pages**
    - PAX implements non-executable VM pages on architectures that do not support the non executable bit. It makes use of /Include/asm-<arch>/processor.h to support executable and non executable pages.
    - If the attacked program was running with different (typically higher) privileges than that of the attacker, then he can elevate his own privilege level (e.g. get a root shell, write to files for which he does not have write access to, etc).
    - Enabling this option will let you choose from various features that prevent the injection and execution of 'foreign' code in a program.
  - PAX_PAGEEXEC: **Paging based non-executable pages**
    - The goal of PAGEEXEC is to implement the non-executable page feature using the paging logic of IA-32 based CPUs. Traditionally page protection

is implemented by using the features of the CPU Memory Management Unit. Unfortunately IA-32 lacks the hardware support for execution protection, i.e., it is not possible to directly mark a page as executable/non-executable in the paging related structures.
        - Eg: If CPU is in user mode then access to non-executable pages will cause page faults which PaX handles.
    - PAX_SEGMEXEC : **Segmentation based non-executable pages**
        - This implementation is based on the segmentation feature of the CPU and has a very small performance impact, however applications will be limited to a 1.5 GB address space instead of the normal 3 GB.
    - PAX_EMUTRAMP : **Emulate trampolines**
        - There are some programs and libraries that for one reason or another attempt to execute special small code snippets from non-executable memory pages. Most notable examples are the signal handler return code generated by the kernel itself and the GCC trampolines.
        - If one enabled CONFIG_PAX_PAGEEXEC or CONFIG_PAX_SEGMEXEC then such programs will no longer work under the kernel.
        - One can use the 'chpax' or 'paxctl' utilities to enable trampoline emulation for the affected programs yet still have the protection provided by the non-executable pages.

- **Address Space Layout Randomization**
    - Many if not most exploit techniques rely on the knowledge of certain addresses in the attacked program. The following options will allow the kernel to apply a certain amount of randomization to specific parts of the program thereby forcing an attacker to guess them in most cases. Any failed guess will most likely crash the attacked program which allows the kernel to detect such attempts and react on them. PaX itself provides no reaction mechanisms, instead it is strongly encouraged that you make use of Nergal's segvguard or grsecurity's built-in crash detection features.
    - Full ASLR can only be bypassed in case of information leak. This is possible due to leakage of information during a crash. This method prevents the user of the program to obtain information about random addresses.

- **Binary Marking**
    - PAX_MPROTECT
        - Enabling this option will prevent programs from
          - changing the executable status of memory pages that were not originally created as executable,
          - making read-only executable pages writable again,
          - creating executable pages from anonymous memory,
          - making read-only-after-relocations (RELRO) data pages writable again.
        - Stops ret-to-libc by checkpointing execution flow changes.

Address Space Layout Randomization(ASLR)

ASLR is an effective mitigation technique. The memory positions where the different segments of the process are allocated in the Virtual memory address (VMA) space are not fixed but random, and so the attackers don't know the address to jump to in case of code injection or return to libc attacks. ASLR relies on keeping secret the virtual memory layout. The randomization must be applied to all the objects of a process, specially to objects marked as executable; otherwise, an attacker can use the non randomized object to bypass ASLR. The basic idea of the ASLR consists in placing the objects randomly in the virtual memory space.

There are several constraints which limit the effectiveness of ASLR:

- **Fragmentation problems using the virtual memory:** The inability to use memory that is free. Although objects are allocated in virtual addresses, once an object is mapped in a position, the application will use the given address to access it, and so it is not possible to move it to another address unless the application is completely aware of it. As a result, VMA will get fragmented if objects are randomly mapped. Eventually, a new object allocation would fail when there is not a free range of virtual memory large enough to hold it.
- **Entropy:** The level of uncertainty that an attacker have about the location of the given object. Less entropy would render ASLR meaningless in 32 as well as 64 bit systems. Entropy gets impacted due to fragmentation which VMA suffers as a result of mapping of several libraries.

As a result of above constraints, several vulnerabilities in ASLR have been found as listed below:

- **Low Entropy:** Linux version limit the growth of stack beyond 8 MB via ulimit facility. The heap structure is used to store malloc/free allocations but when a single block of dynamic memory is requested, a more flexible mechanism to allocate memory is used by the Glibc.The ASLR has been implemented adding some randomness to the original positions of the objects. In order to maintain the relative position of each object, allow

growable objects(stack and heap) and also to avoid fragmentation. As a result, the VMA space used to randomize each object is small with respect to the full VMA. PaX has more entropy than Linux as far as mapping objects are concerned in VMA space. Unfortunately, the largest allocatable object in PaX is only 692 MB, which may break some applications that require a 1GB chunk of memory to be allocated for an object. In 64 bit systems, PaX has lowest entropy on one of the most critical objects, the executable (EXEC).

- **Non uniform distribution**: The libraries and mmaps in 32-bit system in PaX are not uniformly distributed in the available address range.
- **Correlation between maps:** If knowing the position of an object gives information about where another is in memory, then it is said that they are correlated. According to how an attacker can use the correlation, 3 types can be defined: total, partial and useless.
    - **Total correlation:** If knowing the position of an object gives the exact position of another one, then these two objects are said to be "totally correlated". This is the most obvious, known and dangerous correlation and can be found in current ASLR implementations including Linux and PaX. All libs are mapped side by side, and so de-randomizing one lib gives the positions of all the others.
    - **Positive correlation:** When knowing the position of an object gives us enough information to guess another one with less effort than guessing the second one directly. Its called positive correlation, because a leak of an address belonging to an object reveals some information (some bits) about where the other object is mapped in memory.
    - **Useless correlation:** Knowing the position of one object doesn't give any information about the position of another one.
- **Memory Layout inheritance**: It is widely known that children process inherit/share the memory layout of the parent. Child process will know the position of future mappings of all other sibling processes, as far as ASLR algorithm is deterministic.

In order to provide mitigation against above vulnerabilities, an idea of ASLR-NG has been proposed as mentioned in [11]. Below are the 2 important aspects of ASLR-NG:

- **Addressing the fragmentation issue:** When the process is initialized, the upper of lower half side of the VMA is chosen at random (one random bit); and then during the execution of the process, the allocation algorithm uses the selected part to randomly allocate the new objects. This 2 phase randomization method provides both full range randomization and at the same time half of the VMA is reserved to attend large requests.
- **Limiting page table size:** Another problem that must be addressed is the size of the page table. Even limiting the VMA used to randomize every object and mmap request, the page table size may be enormous, compared with current ASLR designs.

More details about ASLR vulnerabilities and ASLR-NG can be found in [11].

Thus, it can be summarized from the above vulnerabilities of ASLR that PaX which completely depends on ASLR may not provide adequate security for docker in the near future. The

container security can be revisited when ASLR-NG or anything equivalent has been implemented as part of PaX systems.

SELinux

What is SELinux?

SELinux is an access control mechanism that is used to enforce fine-grained policy and type enforcement using Labels. In other words, SELinux is a Labeling system, where every process, file, directory and system object has a label. SELinux allows policy rules to be created, which control the access between the Labeled processes and Labeled objects. The kernel acts as a monitor and enforces the rules that the user has created.

SELinux is based on United States Department of Defense Style Mandatory Access Control System (MAC) that provides an enhanced mechanism to enforce the separation of information based on confidentiality and integrity requirement. In contrast, standard Linux is based on Discretionary Access Control (DAC) which makes use of file modes (-rwx-rwx-rwx). These file modes are modifiable by the owner of the file and the applications that the user is running. Hence, the user application, configuration and implementation need to be correct and without vulnerability else compromise to any one of these would make the entire system vulnerable based on the privileges the user was running on. Another additional feature that SELinux provides compared to the DAC is that SELinux allows one to specify access to resources other than files. Eg: we can have policies to restrict access to network resources and inter-process communication (IPC).

Why is SELinux not used widely?

The policy language that SELinux uses is extremely complex, which is one of the main reasons it is not widely used in the industry. SELinux relies on correct labels for it to function correctly. Hence, whenever a new file/Directory is created it is important to keep them under the correct label with proper granularity, else the applications would not work. Due to these complexities and lack of understanding of custom SELinux configuration, SELinux is not easily accepted even among the security-conscious administrators.

Linux distribution support?

Many Linux distributions now provide support for SELinux. It is supported by Red Hat Enterprise Linux (RHEL) , CentOS , Fedora 2+, CoreOS. Support is also provided by Google Android in which it is called SE-Android. In terms of container environment, LXC, Docker and Rkt support the SELinux implementation. Using SELinux with Docker is very well explained in Docker SELinux security policy by Red Hat in this link.

Working with SELinux

SELinux is enabled by default in Red Hat and Fedora. In order to check if SELinux is enabled and actively enforcing the policies we can run “getenforce”

```
[root@localhost ~]# getenforce
Enforcing
```

If the above command returns otherwise, that would mean SELinux is disabled. A major reason SELinux is disabled due to its complexity and the issues it creates. A easy solution most system administrators take is disabling it. Well, disabling is not a solution. Most of the issue that arise with SELinux is due to improper labeling**.** SELinux has a complex implementation but it is simple to understand. Learning how to customize the labels based on the system being used would help prevent improper labeling. How to customize the SELinux labels would be discussed in detail; in the later sections of this report.

Labels:

SELinux is a Labeling system. Each file in the system has a label of the form:

User:role:type:level

Eg:

Apache server file label

system_u:system_r:httpd_t:s0

system_u:object_r:httpd_sys_rw_content_t:s0

For containers environment, the focus is more on the level, where similar containers having similar types and security levels would still have different labels using categories. The category field is included along with other 4 labels. This is used to provide Multi Category Security.

Eg:

```
system_u:system_r:svirt_lxc_net_t:s0:c186,c641
```

```
^      ^           ^          ^     ^--- unique category
|      |           |          |----  secret-level 0
|      |           |--- a shared type
|      |---SELinux role
|------ SELinux user
```

(diag ref: https://access.redhat.com/)

SELinux is a type enforced system. Hence it differentiates processes from each other using types. Here, we would focus on the type field and the level field.

SELinux supports two forms of MAC:

1. **Type Enforcement** – Represents the domains in which the process would run. The policies created make use of type to control the access. A process from one type cannot access resources from another type.
2. **Multi-Level Security** – Similar to Bell-La Padula (BLP) model, it has different levels of access, where restricted information is separated from classified information to maintain confidentiality. (In multi-tenant environment).The MLS labels define a sensitivity level (s0-s15) and category of the data (c0.c1023)

The MLS / MCS services are now more generally used to maintain application separation, for example SELinux enabled:

§ MCS categories is being used by virtual machines to allow each VM to run within its own domain to isolate VMs from each other

In other words, Type enforcement protects host from processes. MCS protects the one process from another process.

Some of the common issues faced when using SELinux are identified and possible solutions are explained below.

Issues:

- **Every process and object has a label, if not labelled correctly, access is denied:** SELinux works on labels. There is a pretty high chance of labels being incorrect when we say that SELinux doesn't work properly and access is denied. Checking the labels where the permission issue is visible should be the first step in debugging." **Ls –Z /path"** tells you what the label is for that particular file.
- **Alternative paths for confined domains, SELinux needs to know:** This is a common issue where we make mistake. We change the path for certain files belonging to a particular domain but we do not add them in their expected path. Hence, SELinux doesn't find the matching label when an attempt to access those files is made and an error is thrown out. The solution to this problem is to use **"semanage"** and **"restorcon".** Eg: storing apache http files in "/srv/myweb" instead of "/var/www/html"

    # semanage fcontext –a –t httpd_sys_content_t '/srv/myweb(/.*)?'

    # restorecon –R /srv/myweb

Semanage fcontext command is used to change default settings.

Restorecon command used to apply default label to file system objects.

Here, semanage will apply the label to the directory '/srv/myweb'. Restorecon takes the labels and apply it to the inode. It also applies system default configurations to the objects.

- **Want every file in a directory to be labeled in a particular way:** In many scenarios, a user wants to use an alternative path to store files and wants the correct labels to be applied to those. It would be a tedious task to run the semanage command on every file in the directory. A solution to this is using File equivalence. It labels the entire directory to be same in terms of labels as other directory. Eg: A user moves its Home directory to /u/home instead of /home.

#semanage fcontext –a –t home_root_t '/u(/.*)?'

#semanage fcontext –a –e /u/home /home

These commands will use the label "home_root_t" for all the file under the directory /u and make the entire /u/home directory equivalent to /home directory and label as if they were under /home.

- **Moving directories between user space and system space:** When we have SELinux enabled, every resource has a label assigned to it. When we use the move command to move particular files from one directory to another, the labels are preserved. Hence we have the old labels on the files in the new location. When we try to access those files, we would get a permission denied error as SELinux thinks that the labels do not match. Hence, it would not allow the access to those files. The solution here would be to use the "restorcon" command in that directory where the files are moved and it will correct the labels for the files.
- **Building a policy module:** There are times when it is simple to build our own policy and apply it to the files. There is a feature called audit2allow that helps with this. Audit2allow determines the policy that needs to be applied and that are enforced using 'semodule' command

 # grep postgresql /var/log/audit/audit.log | audit2allow –M mypostgresql

#semodule –I mypostgresql.pp

The policy looks similar to the one shown below.

Allow unconfined_t ext_gateway_t : process transition

The policy states that a process running in the unconfined_t domain has permission to transition a process to the ext_gateway_t domain. However it could be that the policy writer wants to constrain this further and state that this can only happen if the role of the source domain is the same as the role of the target domain. To achieve this a constraint can be imposed using a constraint statement:

Constrain process transition (r1 == r2) ;

Takeaway:

- SELinux are access control mechanism using labels and every resource in the system would have a label assigned to it.

- Majority of the SELinux issues arise due to incorrect labeling. Label them correctly
- SELinux is simple if we understand how it works. Customizing our system to have correct labels in all the directory would provide an additional layer of security to the files compared to the discretionary access control that linux provides by default.
- **Do not disable SELinux.**

Conclusion

As part of this project, we performed extensive analysis of security aspects of docker containers. We looked into five major profiles pertaining to security of containers: Seccomp, Capabilities, GRSec, PaX and SELinux. As part of seccomp profiles, we analyzed 270+ system calls and identified threats into three categories - information leakage, privilege escalation and denial of service. We also grouped system calls based on five categories - process control, file management, device management, information maintenance and communication. Based on these two categories we designed and automated the procedure for creating custom seccomp profiles.

We analyzed 14 linux capabilities (which were whitelisted by docker) and came up with a set of dangerous capabilities which when exploited might result in significant damage. We also divided these 14 capabilities into three categories - Network Communication, File System and Process Control which will help developers to add/drop them based on application running on containers.

As part of GRSec and PaX, we identified several best practices which needs to be followed in order to enhance container security. A brief description of role based access system followed by a description of sample policy configuration file was provided. This was followed by a brief overview about PaX where we mentioned about three important characteristics of PaX patches - executable space randomization, binary marking and ASLR. We also identified several vulnerabilities which were found recently in ASLR and the mitigation techniques which have been proposed for the same.

Finally, as part of SELinux we provided a brief description of its working. We identified several issues like incorrect labeling which might result in compromising docker container security. Then we concluded by asserting that although SELinux is complex for majority of developers, it would be best if we don't disable SELinux.

Future Work

- As part of seccomp analysis, we categorized system calls based on their functionality and on the threats encountered. Based on this categorization we created separate

seccomp profiles. But this categorization may not be extensive since several sub-categories might be needed before seccomp profile can be generated. For instance, in network specific syscalls, it might be advantageous to have sub-categories like managing icmp traffic, managing dns services, etc which might help developers in taking more extensive hardening measures for docker containers and would eventually cover more application scenarios.

- An API for seccomp profile generation might be helpful in expediting the process of generating seccomp profiles in DevOps environment. For example, if we have a python API by the name of seccomp.py in which we define all the functions for creation of seccomp, then all developers need to do is to import 'seccomp' in their python script and use the in-built corresponding functions directly along with their arguments.
- Several important claims were made in the section on GRSec and PaX: having a policy configuration file with soft and hard limits defined would help to mitigate denial of service attacks, role based access system would help in minimizing the damage caused due to privilege escalation attacks and ASLR-NG in PaX systems would help developers in keeping randomness intact when libraries are mapped onto virtual memory address space. All these claims can only be tested with patches available on grsecurity website but these patches are for customers only. It's necessary to test the claims made above with these patches which would allow enterprises to make significant strides in container security.
- Given that SELinux is a complex topic and it requires many parameters for proper configuration, it would be beneficial to have a deployment guide in which every aspect of SELinux along with its configuration details is clearly mentioned. This would help in easing out the deployment of SELinux in production environment and will result in enhancing container security to a great extent.
- OpenVZ is another container based virtualization technology available for linux which claims to create secure, isolated linux containers on a single physical host. Although docker is much more popular than openVZ but it would always be more advantageous to an enterprise to have multiple secure containerized technology at their disposal. Hence it would be beneficial to perform security analysis of openVZ containers as well which might give enterprise more options to choose and eventually help them in deploying containers in their production environment.

References

1. https://www.centos.org/docs/5/html/Deployment_Guide-en-US/selg-overview.html
2. https://wiki.archlinux.org/index.php/grsecurity#RBAC
3. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=759c01142a5d0f364a462346168a56de28a80f52
4. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2148
5. https://en.wikipedia.org/wiki/Security-Enhanced_Linux
6. NCC Group Whitepaper Understanding and Hardening Linux Containers
7. http://danwalsh.livejournal.com/30565.html
8. https://selinuxproject.org/
9. http://cecs.wright.edu/~pmateti/Courses/7900/Lectures/Security/NSA-SE-Android/Figs/selinux%20architecture.png
10. https://access.redhat.com/
11. https://www.blackhat.com/docs/asia-16/materials/asia-16-Marco-Gisbert-Exploiting-Linux-And-PaX-ASLRS-Weaknesses-On-32-And-64-Bit-Systems-wp.pdf
12. https://github.com/docker/docker/issues/22252
13. nvd.nist.gov/nvd.cfm?cvename=CVE-2002-0973
14. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0145
15. https://docs.docker.com/engine/security/security/
16. https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf
17. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/SELinux_Users_and_Administrators_Guide/
18. https://grsecurity.net/
19. https://pax.grsecurity.net/
20. http://man7.org/linux/man-pages/man7/capabilities.7.html