Secure Operating Systems through Design

J.L. van den Berg
jbg930, 2539148

Abstract—The abstract goes here.

I. Introduction

Operating systems are programs or collections of programs that together form software that is the foundation of many computer systems. Their purpose is to handle the complex task of managing the various resources, such as hardware (such as memory) and software (such as processes), as visualized in figure 1. They act as an intermediary for other applications to receive and access these resources and provide primitives and services for other applications to use, providing an abstraction layer that removes the need to work around hardware specifics and more on providing actual functionality. While there isn't a clear notion of where the boundaries of where the operating system ends and normal applications begin lies, there is generally a single process on the computer system that is running at all times, the kernel. This kernel usually runs using elevated privileges compared to user processes, making sure that normal applications cannot interfere with it or other processes.

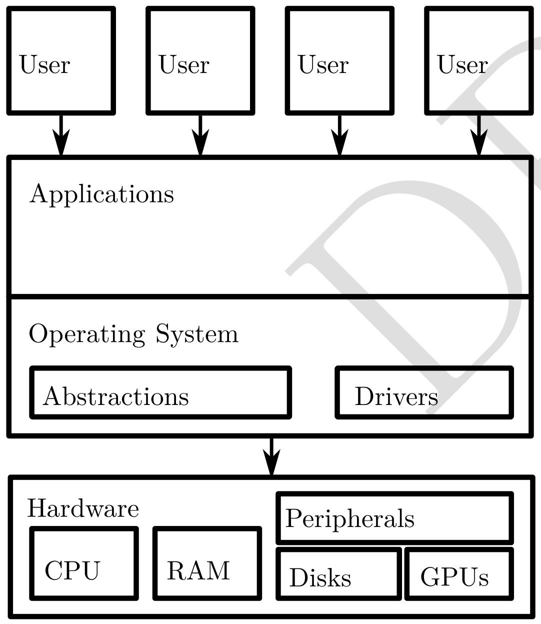

Fig. 1. Structure of Operating System with regards to hardware and applications

In this literature review we will answer the question which tactics are most widely used by operating systems to protect against malicious threats. To do this, we will provide a non-exhaustive list of past- and present day operating system defences and compare the types of attacks these defences will protect against.

II. Background

A. Processor

Arguably the most important component of any computer system is its Central Processing Unit (CPU). Its job is to fetch an instruction from memory, decode it, execute it, and repeat the cycle until there is nothing left to do. This dictates the way that programs are executed. Each processor has a set of available instructions it can execute, called the processors architecture, such as Intel x86, ARM, and AVR.

Communication from the CPU to system memory is relatively slow. To mitigate this, there are several memory slots called registers inside the CPU. Most of the instructions of the CPU will use these registers as arguments. There are several types of registers, where the most common is the General Purpose Registers (GPRs) which can store both data and memory addresses for instructions to use. Besides the GPRs there are also Special Purpose Registers (SPRs), such as the Program Counter, which keeps track of what instruction to execute next, the Stack Pointer, which keeps track of the current stack frame that is being operated on, a flag register, which keeps track of several flags such as results from comparisons or effects of instructions.

B. Memory

The second most important part of any computer system is its memory. Memory consists of an array of individually addressable bytes, the amount ranging from a few bytes up to hundreds of gigabytes. It is the only storage the CPU has access to. Other storage is required to first copy data into memory before the CPU can use it. The notion of memory usually refers to main memory, usually called Random Access Memory (RAM) which contains the data and instructions that are needed in process execution by the CPU. RAM is normally implemented using volatile types of memory causing the information that is stored to be lost when powered down. Besides RAM, many systems also contain a small amount of non-volatile Read Only Memory (ROM) which can, for example, be used to load the bootstrapping code from at system boot.

While we ideally would have memory that is as fast and as big as possible, this is not possible due to the prohibitive costs this would entail. Instead, modern computer systems consists of a hierarchy of memory, where memory becomes more scarce the faster it becomes. At the pinnacle of this hierarchy sit the CPU registers, which are as fast as the CPU itself but are extremely limited in capacity. These registers are generally handled by the software itself.

Below registers we have cache memory. Caches remain small, but are much faster than main memory and are used to hold recently, or frequently used data, and is mostly controlled by hardware. Accessing these caches only takes a few clock cycles, whereas accessing RAM takes a handful of nanoseconds [1].

Aside from the main components listed here, a modern processor contains many more parts, such as the Memory Management Unit (MMU) and the Translation Lookaside Buffer (TLB) that we will discuss in later sections, others fall outside the scope of the literature study, but should be taken into account by the OS to achieve true security, as almost every component can be exploited somehow [2]–[4].

C. I/O and drivers

Computer Systems generally have some kind of in or outputs attached to them, such as disks, mice, keyboards, networks, etc. It is one of the operating system's jobs to manage all these devices and provide uniform ways to applications to access these devices. It does so by providing generic interfaces to deal with categories of devices, as well as low level primitives for interacting with things such as ports or busses. The problem with providing generic interfaces is that there can be massive differences between the various devices not covered by this interface, that need specific code to handle its quirks. To deal with these, operating systems generally provide a way for hardware vendors to provide drivers, special programs that can convert the generic I/O calls into the relevant calls to lower level primitives. How these drivers are implemented is wholly dependent on the OS. As such, decisions such as if it is loaded statically or dynamically, provided with the OS or separately, and whether or not its ran in kernel- or userspace are decided by the OS itself.

D. Processes and Threads

When a program is executed, the operating system will wrap it in a process, a representation of the state of the program during its execution, allowing it to keep track of its state during execution. Associated with each process is an address space for that process alone. In this address space the program text, data, and stack are loaded. Besides the address space, the process also contains all information related to the current state of the process, such as the program counter, stack pointer, other CPU registers, and open files and current positions in these files.

Should the OS need to suspend execution of a process at a given time, it can store the current state in the process. When the OS is ready to resume execution, it can reload the program state from the process and restart the program, transparent to the running program itself. This save and reload cycle allows the OS to run many programs 'at the same time' on a single processor core by assigning each process a slice of the available processor time. The OS can also swap out a process if it has engaged some sort of IO. These operations take a relatively long amount of time to complete, and as such the time waiting could be better spent working on something else.

Should a single process want to execute several functions at once, the process could spawn a secondary process. These new so-called child processes will then receive their own address space, program counter, and register set separate from the original parent process. However, it is sometimes convenient to have several interactions happen at the same time while explicitly sharing the address space, allowing the interactions to work together without inter-process communication. We call this concept multithreading, and each individual flow of control a thread. These threads share an address space, code, and data segments, but each have their own set of registers and stack.

E. Filesystems

Not all workload's storage needs can be met by solely relying on RAM. Some workloads can require large amounts of storage, larger than the amount of RAM equipped on the system. Other workloads might require the stored data to be accessible across system restarts. One solution for these problems lies in using external storage devices, such as hard disk drives (HDDs) or solid state disks (SSDs).

While writing data directly to the disks is a valid solution to this problem, Operating Systems generally provide more user friendly abstractions built upon the disks. Partitions allow the operating system to subdivide the disk into several regions that can each be managed separately.

Filesystems provide a user friendly abstraction to actually storing data on disks, and generally consist of two primary concepts: files and the directory structure. Files are the operating systems structure for the storage of a single piece of self-contained data on the file system, and are typically named. Its name is supposed to be a unique, usually human readable, identifier for storing and retrieving the data from the disk, and is detached from the actual position the file is stored at on the disk. Besides uniqueness, rules for naming files differ from filesystem to filesystem, such as case sensitivity and length. However, most file names have an extension, a short identifier after the name that specifies the type of data stored within the file. Depending on the filesystem or operating system, this could trigger different behaviour depending on the file, but it could also just serve as a convention reminding the user what type of data is contained inside the file. Besides the name, file systems tend to store additional metadata such as the size and timestamps such as when the file was created.

Besides regular files, which contain actual data, there are also several special types of files, such as directories.

Directories form the basis of the file structure of the file system. While old filesystems might have had just a very simple structure, having just a single directory called the root directory, more modern filesystems have a hierarchical structure, where you still have the root directory, but allow for many subdirectories and files. This allows the user to arbitrarily group files in a way that makes sense for the workload, and for providing access controls as we will see later in section III.

F. Ground Principles

When building any secure system it is useful to have a set of agreed upon design principles to guide the development of its components. While such principles might not guarantee the resulting system to actually be secure, it at the very least provides points of discussion to steer the design.

Probably the most well-known set of these principles are the principles formulated by Saltzer and Schroeder, created while they were working on MULTICS (Multiplexed Information and Computing Service). MULTICS was a joint project between MIT's Project MAC, Bell Telephone Laboratories, and General Electric Company's Large Computer Products Division and was started in 1965 until it was ultimately cancelled in 1985. From the beginning, MULTICS was designed to be a multi-user general-purpose computer system [5], while also being the first major operating system to be designed for security from the start [6]. MULTICS introduced various novel ideas and implementations of various OS and security features, including the hierarchical file system, memory mapped files, dynamic linking, and kernel rings.

In 1974, after the project had been in development for a few years, Saltzer produced a technical overview of the various security features present in MULTICS, including a list of design principles [7]. This list was improved and expanded with the help of Schroeder in 1975 [6] and included the following principles:

1) Economy of Mechanism, meaning that the design of the security measures should be as small and simple as possible. The motivation behind this principle is that as a piece of software grows in complexity or size, the larger the chance a subtle weakness or exploitable flaw find itself inside the codebase. Saltzer and Schroeder argue that the access paths that lead to such a flaw will generally not be noticed under normal use of the system, as normal use does not include attempts to access those paths. This necessitates a line-by-line inspection of the software and physical examination of the hardware. Since every path should be inspected to guarantee security of the whole component, a smaller and simpler component is thus easier to test and verify than a larger one.
2) Fail-safe Defaults means that access permissions should be based on permission rather than exclusion. That is, that by default no access should be granted to anything, preventing any unauthorized access. The access scheme is then used to determine the conditions that access should in fact be permitted. This strategy was originally suggested by Glaser in 1965 [8]. Saltzer and Schroeder argue that the alternative (permitting access by default) presents the wrong psychological base for secure software design, and that in large systems some objects will be inadequately considered. The approach of explicitly granting permission limits the damage that can be caused by such inadequately considered objects. Since the failure mode is to reject access, such mistakes can be detected quickly, whereas in the alternative this can go undetected during normal use.
3) Complete Mediation means that every access must be checked against the access control mechanism. This forces a single, system wide, source of truth for all access control. This design requirement also forces the system designers to devise a foolproof method of identifying the source of any access request made to the system. The principle also views caches very sceptically, noting that systems should not rely on the access decisions retrieved from the cache, and that if caching is implemented, careful consideration should be given on how changes in authority are propagated.
4) Open Design means that the implementation of any security mechanism should be public rather than secret. Any such security mechanism should thus decouple their internal protection keys from the implementation. This allows the implementation to be scrutinized by users and experts alike, reinforcing confidence in the implementation, without compromising the security of the mechanism, as the protection keys remain secret. Saltzer and Schroeder also make the point that it is ”simply not realistic to attempt to maintain secrecy for any system which receives wide distribution” [6].
5) Separation of Privilege is described as the practice of securing a system with multiple keys to both increase security and flexibility. When using multiple keys, the keys can be separated from each other and separate entities can be made responsible for their safekeeping. This prevents an accident (or a single rogue actor) from compromising the entire system, as they would require the secondary key held by a different entity. An example of this principle in use is Multifactor Authentication (see section III-B), whereby a single user is identified using several different factors (such as passwords and a phone based token).
6) Least Privilege means that every single program and user on a system should operate using the smallest set of privileges possible while still being able to perform their intended task. Doing this primarily limits the damage that errors and accidents can cause, but can also reduce superfluous interactions

between privileged programs so that unintentional or improper uses of privileges are less likely to occur. Together with principle of fail-safe defaults, this forms the basis of various access control schemes (see section III-C). According to the principles, these should grant no privileges by default, and based on the user, and the requirements of the user, be provided the privileges they require. Under the principle of least privilege, even administrators should not have administrative privileges unless their current task requires it, which is when they would elevate their privileges (such as through sudo). When their work has concluded, they would have to drop down back to their normal privileges.

7) Minimize Common Mechanisms means that the design should minimize common functions shared by all users. This should be done as each shared mechanism represents a potential path of information between users. By minimizing the common components, you thus reduce the number of potential paths. Furthermore, this reduction in common components reduces the code base that would have to be analyzed for security, and allows users that are not content with a mechanism to potentially replace them with another one.
8) Psychological Acceptability means that security mechanisms (and other human facing interfaces) should be designed for ease of use, and to fit the mental model of the user. By following the mental model of the user, it allows users would more frequently apply the security mechanism, while also making fewer mistakes. A mechanism that confuses or hinders the user, may instead cause the user to make more mistakes, or in the worst case, disable the mechanism entirely.

In the 1975 paper by Saltzer and Schroeder [6], they also introduce two more principles based on physical security. These principles were not incorporated in their true list, as they only partially, or imperfectly to computer systems. These principles were:

9) Work Factor is the implied cost of attempting to break a security mechanism compared to the resources that a potential attacker could field. The problem with this principle with regards to computer systems is that while some mechanisms are easy to calculate a work factor for, a lot of mechanisms are not susceptible to such calculations. As an example, it is easy to calculate the amount of work required to guess all passwords of exactly 8 characters in length, and it is also easy to see that increasing the length of passwords, or allowing more characters, dramatically increases the work required to brute force the password. But by the 1970s, Saltzer and Schroeder found that attackers often penetrated systems, not by brute force, but by searching for and exploiting vulnerabilities in implementations. Such attacks are very hard to quantify into a work factor, leading Saltzer and Schroeder to question the principle's relevance on system security.
10) Compromise Recording is the action of logging or otherwise keeping a track of attacks and compromises, even if the actual attack is not blocked. Saltzer and Schroeder are sceptical of the benefit of this, as such a tool might not be able to guarantee discovery of an attack or compromise, and that if a system couldn't prevent an attack that modified data, it could undo the creation of the compromise records.

Later, in 2009, Saltzer and Kaashoek released their book 'Principles of Computer Systems Design: An Introduction' [9]. From this book we can add the last few principles to our list:

- Minimize Secrets: Secrets is the generic term for any information that should remain hidden from an attacker of the system. This includes, but is not limited to, encryption keys, passwords, API tokens, etc. Since these secrets can become compromised, they should be made simple to replace should this eventuality come to pass. The amount, but not the quality of the secrets should be limited to reduce the administrative overhead of having many secrets.
- Adopt Sweeping Simplifications: Since components can become very complex, simplifications can provide a measure of control by making specific details irrelevant, allowing the designers to make more compelling arguments towards their system's correctness.
- Least Astonishment: A more concise and clear restatement of the principle of Psychological Acceptability, this principle focuses not on the goal itself, but on the easier to understand problem.

III. Protection Techniques

A. OS Structure

There are several ways to structure operating systems. The most common way is by means of a Monolithic kernel. In a monolithic kernel, all functionality of the operating system is bundled into a single application. Inside monolithic kernels all functions can call all other functions, which can be more efficient than other structures as there is no need for communication between processes and allows for a lot of freedom by being able to reuse almost any previously written function. They also usually support extensions to the kernel through special programs called kernel modules. These modules must adhere to the interfaces provided by the kernel, but in turn allow the program to run inside the kernel address space. The biggest downside to this approach is that should a crash happen anywhere, for any reason, the entire kernel crashes with it, rendering the system inoperable. The surface allowing such a crash becomes much larger when looking at drivers, as it only takes a single buggy driver to take the entire system down.

Another way to structure an operating system is by doing the polar opposite of the monolithic design, called a microkernel. By putting as little code inside the kernel as possible, they try to reduce the code surface that allows critical bugs and crashes. Microkernels attempt to encapsulate all functionality into small, interconnected modules. Only one of these modules would become the kernel, and all others become normal user processes. This includes drivers, ensuring that a buggy driver cannot bring the whole system down with it.

Next we have the Unikernel approach. Unikernels are constructed by combining several library operating systems, where the components of an operating system, such as processes and networking, are implemented as libraries. By mixing and matching these library routines, you can create a completely customized, tiny operating system to fit your exact needs of the application you want it to run. Unikernels only have a single address space, meaning that there is (almost) no separation between kernel and the application that the OS is built around.

B. Authentication

In any secure system, security relies on knowing the identity of any communicating entities. Authentication is the act of verifying the truth of an attribute, such as an identity, claimed by an entity, such as a user. With this identity the system can perform the act of authorization: specifying an entities privileges and access. Access Control will be expanded on later in section III-C.

When authenticating an entity's claim, there are three factors that can be used to validate their claim:

- Knowledge: This is the most common factor in authentication and relies on the entity having information that only that entity should have. Examples of authentication using this factor are passwords, PINs, and challenge-response mechanisms.
- Ownership: The second factor in authentication relies on the entity possessing something that attests to his claim. Examples of this are smart cards, security certificates, and security tokens, both hardware (such as U2F tokens) and software (such as a phone based HOTP/TOTP generator).
- Inherence: This last factor verifies something that the entity is (such as a fingerprint, retinal scans, or Face ID), or something that the entity does (such as voice patterns, handwriting / typing characteristics).

Authentication can be done using any number of the above factors. The simplest form, single-factor authentication, only uses one factor. Using only a single factor provides the weakest level of security and does not provide much security against malicious entities. Especially the most common form of single-factor authentication, using passwords, is often easily compromised due to bad user password practices and data leaks [10]–[13]. Stronger authentication guarantees can be achieved by using two or more independent factors, such as using password plus a one time password. This process is called multi-factor authentication.

C. Access Control

In the previous section we discussed the need for authentication in a secure system. But besides knowing the entities at play, we also need a way to control what actions each entity is able to perform, i.e. access control with at the very core the Principle of Least Privilege [7].

In Operating Systems, access control is mostly concerned with limiting the activity of legitimate users or processes running on behalf of such users. The OS needs to, for example, validate and control the access to files, memory, and processes so that users cannot access that which they are not allowed to. In operating systems this is achieved through a concept of a reference monitor, an uncircumventable type of interface for programs to communicate their intent with the OS, which allows the OS to control every attempt at accessing such resources.

1) Access Control Policies: Several policies regarding access control exists.

- Discretionary Access Control (DAC): Established in the TCSEC, DAC allows subjects to govern access to the objects they control themselves [14]. This provides a lot of freedom for the subjects, however does not provide a lot of security. Implementations of DAC are usually done by having one or more owners assigned to each object in the system. These owners are free to make changes to the permissions on that file.
- Mandatory Access Control (MAC): Another policy established by the TCSEC, MAC pushes the governing of access to the system itself using a preset security model. While the term started out closely associated with MLS and the Bell-LaPadula model [14], it has started to become more widespread, with modern implementations such as AppArmor and SELinux being actively used today.
- Lattice Based Access Control (LBAC): Label based subclass of MAC that restricts information flow based on a lattice structure [15], [16]. A lattice is used to define the levels of security that an object may have and that a subject may have access to. The subject is only allowed to access an object if the security level of the subject is greater than or equal to that of the object.
- Role Based Access Control (RBAC): Instead of assigning privileges to specific subjects, privileges can be assigned to roles, and these roles are then assigned to subjects instead. Notable improvements over other (older) styles:
  - Users can easily be added to or removed from roles, without affecting any other users or permissions
  - Permissions can be easily added to or removed from a role, immediately affecting every member of that role

  Can be extended to provide hierarchical role inheritance and constraints [17]–[19] Using inheritance and constraints it can create or simulate LBAC models [19], but can also be used to simulate both DAC and MAC [20].

2) Bell-LaPadula Model: The Bell-LaPadula (BLP) model is a formal security model developed by Bell and LaPadula in 1973 under sponsorship of the US Air Force [21] in an effort to formalize their multilevel security policy [14]. The model was later refined and extended for use in the MULTICS system [22], [23].

The BLP model is a multilevel security (MLS) model, meaning that the model divides both subjects and objects into several (incompatible) security levels (such as the US Military's 'Unclassified', 'Confidential', 'Secret', and 'Top Secret'), and allows or denies subjects access to information based on security clearances and the respective object's and subject's levels.

In the BLP model information flow is modeled as a state machine, where the transitions between secure states are only allowed under the following MAC rules:

1) Simple Security Property: A subject at a given security level can only read objects that are at a security level below or equal to his own. This property prevents subjects from reading files that are above their clearance level, and is sometimes also described as 'read down'.
2) * (star) Property: A subject at a given security level can only write to objects that are at a security level equal or higher than his own. This property prevents higher level subjects from leaking information to subjects that are at a lower level, while not preventing lower level subjects from informing higher level subjects. This property is also described as 'write up'.

Together, these rules ensure information can only flow to equal or higher security levels, guaranteeing security. However, there are cases where information should be able to flow downwards. This is why Bell and LaPadula introduced the concept of trusted users in their 1974 revision, which are special subjects that are not bound by the *-property and that are to be vetted to comply with the security policy [22].

The BLP model is designed to keep secrets secure, but does nothing to preserve the integrity of information, as a lower level subject is able to write to all files above his level, therefore possibly feeding intentionally compromised information to higher level subjects. Biba learned however, that by reversing the properties of the BLP model, you were able to construct a model that guaranteed integrity, at the cost of not providing any security [24].

3) Access Control Lists & Capabilities: To achieve access control you need to be able to specify, and store, exactly what actions (e.g. 'read' or 'write') any subject can do on a given object. This is achieved by first assigning every single subject and object a unique identifier. With objects this could be the path or the inode number of a file, a memory address and size of a piece of memory, and so on. Similarly, users can be uniquely identified. In general, access needs to be granted to each subject, or role in case of RBAC, at the discretion of the owner of a given object, a prime example of DAC.

A naive approach of storing these access rights is to create a matrix of $subjects * objects$ and add any allowed actions to each field. It doesn't take much to see that the matrix quickly becomes huge, while generally being very sparsely populated as many objects will generally only be accessible to a limited number of subjects. The solution to this problem is simple, we should only store the non-empty fields and assume that by default, nobody is able to access anything.

The first way of storing this matrix compactly is by using Access Control Lists (ACLs). For every object, an extra ACL is created, which stores, for each subject, the exact privileges that subject has on that given object. Should a subject not be listed in the ACL, it implies that the subject has no access to the file. By using an ACL its easy to enumerate the access that subjects have on any given object. This comes at the cost however, of not easily facilitating checking which objects a single subject has access to.

The second way of storing the matrix is by using Capabilities and Capability lists [25], [26]. Instead of listing the permissions per-object, capabilities opt to list all the objects a given subject has access to. By storing the information at the subject rather than the object, capabilities have the exact inverse advantages of ACLs, that being that its easy to see what objects a subject has access to, but not what subjects have access to a given object.

D. Memory

1) Memory layout / segmentation: Back in the 1940s, systems were designed with the intention of only a single application running on the system at any given time. The program could then use (almost) all system resources, including memory, as it saw fit. For programs that would not fit within memory, some systems provided the programmers with the ability to move blocks of memory over to a secondary storage. All blocks had to be divided and accounted for entirely managed by the programmer, leading to a lot of mental overhead [27].

Aside from this, the need quickly arose for the ability to run multiple programs on the same system. The old way of doing memory management turned out to be insufficient for handling this, as programs could read and (over)write the memory of the other programs running on the system. To prevent interference between programs several solutions were implemented over the years.

Arguably the simplest solution of moving all the memory to secondary storage when swapping programs. While this approach effectively solves the problem, it comes with some massive drawbacks, the largest of which is that the transfer of memory to and from secondary storage is very slow.

Aside from this simplistic solution, other computer systems came up with their own solutions. An example of this is IBM's System/360, which allowed programs to share a single address space, but each program would be loaded into at different starting points (base addresses) in this address space. Loading programs into different areas than they expect caused several problems, not the least of which the fact that programs can, at times, jump to fixed addresses in memory, such as function calls. IBM solved this by using static relocation where the operating system, when it loads the program into memory, rewrites fixed addresses to add the base address. This did not solve the entire problem however, as the operating system did not always have neccessary context to determine whether or not a constant was used as an address or not.

Note

Later this was used as a basis for ASLR, maybe roll into it.

OS/360 also provided a solution to the problem of programs reading addresses that they did not 'own' by writing to them, through what is now called memory protection keys (MPK). Memory was divided into 2KB chunks and each running program would be assigned a memory protection key. When a program wrote to a chunk of memory, this chunk would be marked using that programs protection key. Any read from that memory chunk would then verify the reading program's protection key against the chunks key. If the keys matched the read would proceed as normal, but a mismatch would trap to the operating system, preventing access.

Note

Maybe some 'insight' or 'concequences' or something on this. Or drop, honestly its kinda cool but not super interesting. It does pave the way for segmentation though.

Other computer systems introduced similar protections regarding segmentation, such as MULTICS [5] or the Burroughs B5000 [28], [29]. Rather than use static relocation in combination with MPK to attempt to isolate programs, they introduced a more dynamic addressing and protection scheme. Memory was still divided into chunks, of arbitrary length, called segments. Each segment would be given an unique identifier, and all addresses would consist of a tuple of the segment id and an offset within that segment. The operating system gatekeeps access to any of these segments and would transparently swap segments to and from the disk when physical memory would be near capacity.

Later systems (such as the Ferranti Atlas) attempted to create a system thats abstracted away from the notion of segments [30]. Programs would be assigned their own 'Virtual Memory space' where each address would be specific to the program and be mapped to some 'physical' address by the OS. Like with segmentation, memory was divided into chunks, this time of fixed length and called pages, with corresponding 'slots' for them to be mounted in physical memory called page frames. Unlike with segmentation, the addressing of memory happened transparently, there being no notion of something like a segment to the program. Rather, addresses would be translated by the system to a page and offset, and the page would be mounted and addressed proper, swapping pages as needed.

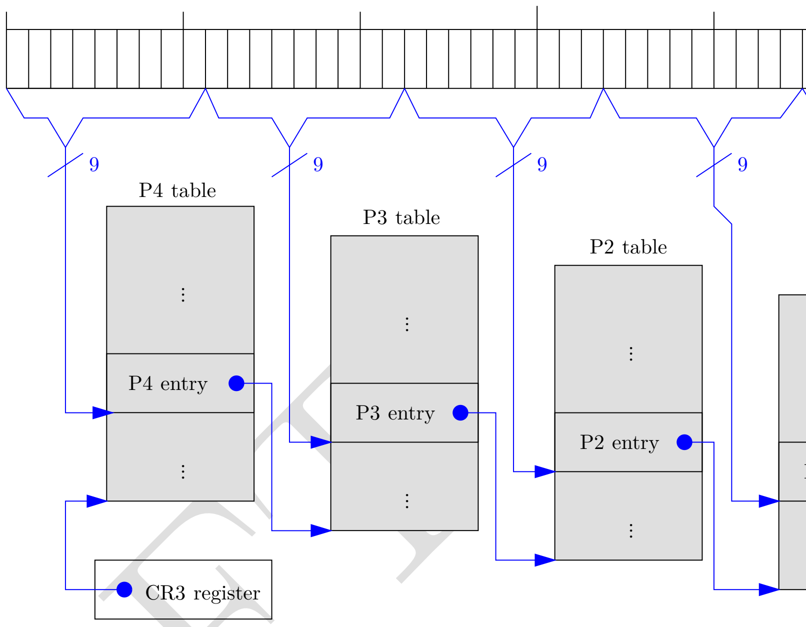

Fig. 2. Multi-level page table lookup on x86 TODO: TEMPORARY UNTIL I MAKE MY OWN CURRENT FROM: https://os.phil-opp.com/page-tables/

The translation of an address to a page is done through an OS managed structure called a Page Table, with systems generally implementing the translation inside a hardware feature called the Memory Management Unit (MMU). These page tables are generally unique to a running program, and when switching between programs, the OS ensures the correct page table is loaded for that program. Since the lookup process and management of the page tables are out of the hands of the running program, this system provides adequate memory isolation between programs. To provide larger memory capacities, modern systems use Multi-Level Page Tables, whereby the a page table can defer the lookup result to another page table, allowing more entries to be stored.

During the lookup, the MMU checks if the address to translate is actually contained within the currently referenced page table. Should the page not be in the page table set, the MMU traps to the OS, allowing the OS to either setup a new page table mapping, or deal with the access violation in another way. If the page was found in the page table, the MMU proceeds with the translation. If the system uses a multi-level page table architecture, this would involve repeating the previous step until the final level. After the translation to an actual page has been completed, the MMU first verifies if the page is already loaded into a page frame, and if so, perform the memory access that the program was intending to do. However, if the page was not yet loaded into memory, the MMU will instead trap back to the OS, to allow the OS to perform the neccessary page swap.

2) Other features: Aside from general memory management techniques as described in the previous section, several additional features for memory security were created over the years, some of which we will touch on briefly.

Executable Space Protection (ESP) is a feature that allows regions of memory to be marked with special policies. When properly used, this defence can protect against some variations of malicious attacks, such as shellcode being injected through buffer overflows (if the stack and heap are marked as not-executable). In its simplest form, this could work very much as the permissions set files, marking some memory as, for example, non-writable or non-executable. An early example of this can be found in the Burroughs B5000, where each word of memory had a single bit of metadata designating it as either code or data. The hardware of the system would enforce the seperation of code and data in such a way, that trying to read from a word marked as code, or executing a word marked as data, would not be allowed [28]. In more modern times, x86 processors have built in support for ESP through the NX bit, allowing pages to be marked as non-executable.

An example of ESP being used is through a memory policy called 'Write XOR Execute' (W^X) which was introduced originally by OpenBSD in 2003 [31]. With this policy the OS enforced that every page (for a given process or the kernel itself) could either be marked as writable or executable, but never both. This policy provides a strong guarantee against both modification of executable code and execution of modifiable data. However, this policy can also interfere with legitimate use-cases, such as just-in-time (JIT) compilation of interpreted languages.

Another feature implemented by many systems is the concept of protection rings, first implemented by MULTICS [5], [32]. Protection rings are based on the observation that hardware permissions in the system tend to be hierarchical in nature. As such protection rings model that hierarchy with the kernel at the pinnacle, being the most privileged code to run on the system, with userspace being the least privileged. Enforcement of these rings is done through hardware (on x86 this is done through the CPL bit of the CS register) with predefined 'gates' being used to allow structured access to functions inside more privileged rings.

While MULTICS historically had 8 different protection rings available to the OS, x86 distilled them to only 4:

0 Kernel
1 Device Drivers
2 User programs with privileged code
3 Normal userspace programs

Even with the limited set of rings available, most systems only used two rings in practice: ring 0 for the kernel and ring 3 for userspace. However, with the development of virtualization, it became commonplace to run the kernel inside a virtual machine. This lead to an unofficial new ring level, ring -1, being used to refer to the hypervisor the system is running on. Some more modern architectures, such as ARMv7, actually encoded this development into the protection rings system themselves, with ARMv7 three rings being:

0 Application
1 Operating System
2 Hypervisor

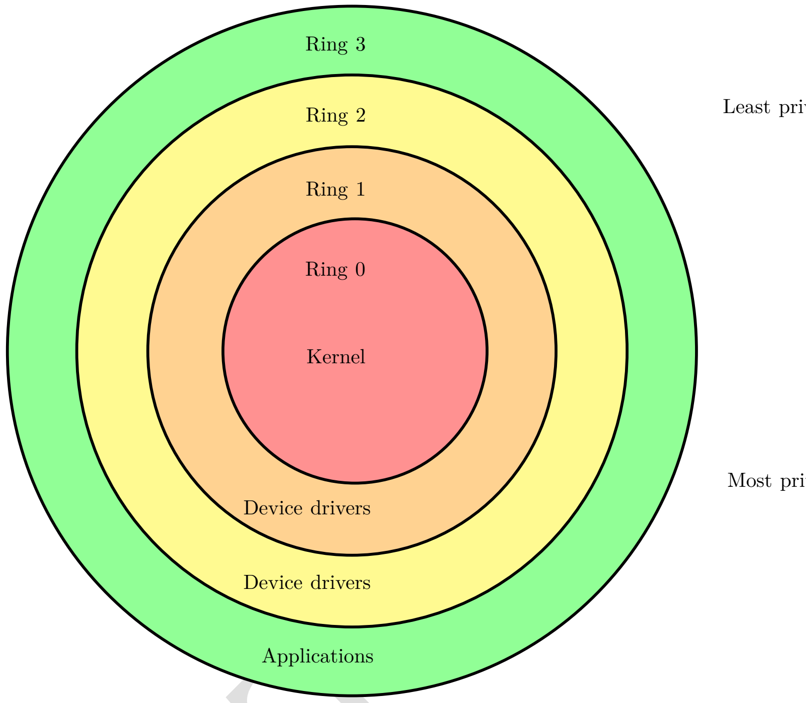

Fig. 3. Protection rings on x86 TODO: TEMPORARY UNTIL I MAKE MY OWN CURRENT FROM: https://en.wikipedia.org/wiki/Protection_ring

Another way that the operating system seperates privileges between the kernel and userspace is through a supervisor mode. Supervisor mode is flag on threads that control the ability of that thread can limit the instructions available to userspace proram, limit the ability to control system interrupts, etc.

This later got extended by Intel with their broadwell architecture, introducing memory protections based on this supervisor mode flag. These protections are Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP) [33], [34]. These two protections combined help protect against accidental execution of untrusted code while running under supervisor mode [35]. While SMEP and SMAP are active, the operating system can mark every page as being in either supervisor mode or user mode, and the processor will enforce that supervisor code can't execute code (SMEP) or read memory from (SMAP) from user pages, and vice versa.

The final defence we will discuss here is Address Space Layout Randomization (ASLR), whereby the locations of critical areas of a program are randomized every time the program is run using static relocation [36], [37]. By randomizing the location of the initial stack, the heap, and the libraries, ASLR helps defend against control flow subversion attacks by forcing attacks to first determine the location of their targets, rather than having the target be knowable beforehand. While usermode ASLR was introduced by the PaX project for linux in 2001, ASLR for the kernel itself was introduced in a very limited form, with later research allowing kernel ASLR in full [38].

E. File system

HJB

I dont think you need much here. It is mostly about access control, which you already covered, and encryption

As we described in Section II-E, one of the various tasks the operating system manages for applications is the management of and providing access to files on various storage media. As with any other part of the operating system, attention is given to several security features of how operating systems interact with the files on-disk.

Usually when files are stored on a storage device they are stored according to the specification and rules of the given devices given file system with which it has been formatted. Access to these files to applications is granted using the OS provided abstractions to do so, such as fopen on linux and the OpenFile or CreateFile functions on windows. This layer of abstraction allows the operating system to provide access control for any application attempting to access the file system, while allowing the application the freedom to not worry about the underlying intricacies of the file systems it might encounter. This access control is generally done using either ACLs or Capabilities as we discussed earlier in Section III-C, and usually expresses these permissions in regards to the user and group the process is running as. This however, means that for the security of the file systems sake, the security guarantees can only be as strong as the guarantees provided with regards to authentication.

These file systems generally have built in storage metadata for storing some of this security related metadata near the file on-disk. An example of this are the extended attributes, xattrs, on the ext4 file system, where ext4 provides space for storing ACLs [39]. However, on Unix-like systems the main way that permissions for files are interacted with are through a simpler interface, the unix permissions model. In this model, files are assigned to a given user and group. Permissions are then managed for the three distinct scopes of user, group, and others.

Each of these scopes has 3 different options: read, write, and execute. Read provides you with the permission to read the contents of the file, the write permission provides you with the permission to change the contents of the file, and the execute permission allows you to execute the file. In the context of directories however, each permission gets applied a little different. The read permission becomes the ability to read the file names inside the directory, but does not provide you with any other information or metadata. The write permission becomes the ability to change this list of file names, by creating, deleting, or renaming entries. Lastly, the execute permission allows you to access the metadata and file contents of a file in the directory, given you know the filename (through the read permission).

Note

Add something about SElinux

1) Integrity: Another facet of file system security is integrity. While not immediately apparent as being needed for security purposes, integrity allows the detection and recovery of data that has possibly been changed by an attacker.

The first of the methods we will discuss here is checksumming. Checksums is a (small) amount of data that is derived from the contents of a (bigger) block of data for the purpose of detecting modification, as a change in the contents of the original block would trigger the checksum to change, and a change in the checksum would mean that verification would fail as the data would generate the original checksum. While originally mainly used for detecting errors during the transmission of data from one system to another, this gradually expanded to using these checksums for verifying the integrity of stored data [40]. An example of checksums in use can be seen in ZFS, where checksums are used to detect read or write errors, allowing ZFS to recovery by attempting to read a redundant copy of the data [41].

2) Confidentiality: Aside from authentication and integrity, the last facet for security on file systems we will discuss here is confidentiality. Confidentiality is the act of ensuring that information can only be read by the intended audience of said file, by only allowing the holders of a secret, such as a password or key, to read the contents of the file in process called encryption. This requirement of needing to hold a secret places importance on key management, as even the strongest encryption will not protect against the leak of a secret key. Due to this, using passwords for encrypting files is not a recommended strategy, preferring the use of keyfiles instead [42]. These keyfiles, besides being stored on the machine itself (such as is common for SSH keyfiles), can also be stored on an external device, such as a USB stick, for added security.

There are two main strategies, or granularities, for encryption:

1) File Based Encryption, whereby the contents of a specific file, but the overlaying structure of the filesystem is not. This allows certain metadata such as paths and at times file sizes to be determined by an attacker.
2) Disk Based Encryption, whereby an entire file system is encrypted. This strategy usually exposes an 'unencrypted' view of the filesystem after the correct key has been given, allowing the user to work with the file system as normal, while all data and changes are transparently encrypted or decrypted.

Irrespective of the strategy chosen, the algorithms used for the encryption remain more or less the same, with most encryption schemas choosing a symmetric cipher, such as AES or Triple-DES to encrypt the data in fixed chunks of data called blocks. However, this does not exclude the possibility of having the secret key stored using an asymetric cipher allowing multiple independent users to access the secret key without sharing their own personal keys [42].

There are several problems with symetric cryptography for encryption, such as that most block ciphers are vulnerable to cryptanalysis or vulnerabilities such as known plaintext attacks or double encryptions [42].

One solution to this problem is by making the encryption more resiliant to attacks by using a process called Cipher Block Chaining (CBC). When using CBC, the encrypted output of a block is also used for the next block, ensuring that any given block depends on the output of all blocks before it. Doing this prevents, for example, two identical plaintext blocks to encrypt to the same ciphertext, preventing cryptanalysis that depends on this feature.

However, this solution does come with its own set of problems, most notably performance. While security is enhanced by the chaining of blocks, any change in a block requires the complete re-encryption of all blocks that followed that block. A single change at the beginning of the file would thus require the entire file to be reencrypted which can become troublesome when dealing with very large files. While this can be mitigated by limiting the total length of cipherblock chains, careful consideration has to be done when deciding this maximum length, as making the chain too short would not provide enough security against the aforementioned cryptanalysis vulnerabilities.

F. Isolation

HJB

I would consider dropping virtualization and containers

1) OS Level Virtualization: Operating Systems level virtualization, also known as virtualization, is the process of running several ‘virtual’ machines on a single system, abstracting away all the hardware, like the CPU, memory, disks, etc. leaving the Virtual Machines (VMs) with the illusion that they themselves are running on their own hardware, completely oblivious to the host machine they are running on. These hardware interfaces are created and managed by the Virtual Machine Monitor (VMM), also known as a hypervisor.

In 1973 Goldberg introduced several formal requirements for OS level virtualization [43]. This was later extended with the help of Popek in 1974 [44]. Amongst other things, these papers laid out several defining characteristics of VMMs.

The first of which is that the VMM should provide an ‘essentially identical’ environment to virtual machine it is running, whereby any program behave exactly as if it were run on bare hardware. They do specify that this explicitly excludes behaviour differences due to resource availability and different timings.

Secondly, the VMM should be efficient, where most of the guest’s code should be run directly on the processor, without VMM intervention. This disqualified most of the already existing emulators and interpreters from falling under umbrella of VMMs.

Finally, the VMM should be in full control of the resources made available for virtualization, meaning that guests should not be able to access any resource not explicitly allocated to it, and that the hypervisor should be able to reclaim resources it had previously allocated to guests.

The papers from Goldberg also specify a distinction between two different classes of hypervisors. The first of which are the type 1 hypervisors, also known as bare-metal hypervisors. These are the hypervisors that run directly on top of the host hardware and are special-purpose built to manage VMMs. As bare-metal hypervisors cannot rely on any underlying operating system, they have to implement several OS primitives, such as memory management, themselves. However, since the amount of these primitives can be limited, type 1 hypervisors tend to be pretty lean.

The other type ov hypervisors are the type 2 hypervisors, also known as hosted hypervisors. These run on a (regular) operating system like a normal application, giving them access to all the abstractions provided by the host OS to manage the VM’s.

Running virtualized systems has several advantages. One such advantage is that the host, and every guest, is isolated and protected from every (other) guest. In effect this means that a failure in a guest does not bring down the host or any guest. Most likely, none of the guests will ever even know that something happened in another guest. This isolation also improves security, as a single compromised system does not compromise the entire host.

Another advantage of virtual machines is on the management side of VMs. A common feature between all hypervisors is to suspend an entire VM. This behaves similar to the suspension of a process on the OS level, whereby the VM does not get assigned any CPU time by the scheduler, leaving it untouched in the current state until the VM is resumed. During this time, hypervisors commonly allow administrators to create a snapshot of the VM. This creates a hard copy of the entire state of the virtual machine, allowing you to reset to that exact state at any point in the future. This copy could also be used to create a new instance of the VM, creating a clone of the original. Lastly this copy could also be used to migrate the guest VM to another host, with modern hypervisors frequently allowing you to do this with (almost) no downtime.

There are several different ways that virtualization has been implemented over the years. The first approach would be using an interpreter, similar to how an interpreter is used for several programming languages. In this style, the hypervisor has to inspect every instruction that the guest would execute and act accordingly. While some instructions can be executed without any intervention, other instructions have special behaviour and have to be rewritten to prevent damage to the host, such as when dealing with (page table) interrupts, while making the guest believe that the changes that it has requested have been made successfully. As you can imagine, the performance overhead of having to inspect every instruction and reinterpret them is very large,

Note

Find a nice source for this, shouldn't be hard

and as such this style is not used anymore for VMMs.

The second approach to virtualization is by using a strategy called trap-and-emulate. The idea behind this approach is that the guest OS will run instructions in user mode (while being told it is in kernel mode) until it reaches a privileged instruction (that is, an instruction that cannot be executed in user mode [44]). These privileged instructions would then trap to the kernel, where the hypervisor can take over and parse and emulate the required response. The problem with this strategy for x86 is that there exists a second class of instructions, called sensitive instructions, where they behave differently in user mode than in kernel mode [44]. While it is immediately clear that all privileged instructions are also sensitive instructions, the problem lies in that x86 has several sensitive instructions that do not trap, and their behaviour cannot be changed for backwards compatibility reasons. Without 'fixing' these instructions, x86 could not be emulated using trap-and-emulate.

This problem was solved in 2005 by Intel when they released their virtualization extension, VT-x [45] and later AMD when they released SVM (now AMD-V) in 2006. These extensions allow the host to create guest 'containers' in which guest OS's can be run, whereby the behaviour of the sensitive instructions is emulated properly by the CPU for kernel mode. All privileged instructions will still trap to the hypervisor for emulation.

A third approach is using binary translation. This is an old and highly complex approach to virtualization which was used by VMware before proper emulation of x86 became possible using VT-x and SVM. This strategy revolves around dynamically rewriting parts of the guest operating system to work around problematic code using pre-existing solutions such as protection rings and multilevel page tables. In cases where hypervisor interaction would be required, the code would still trap towards the hypervisor to be handled in a similar style to trap-and-emulate.

The last strategy we will discuss here is paravirtualization. With paravirtualization the hypervisor does not pretend to be a bare-metal machine and instead provides ways for the guest operating system to work with the host operating system to achieve their goals. This requires the guest operating system to be aware of, and to have support for interacting with, the host operating system using the interfaces they provide. When privileged actions, such as changing page tables, are required by the guest, the host provides a way for the guest to execute a hypercall to affect these changes.

2) Process Level Isolation: Besides isolation on the operating system level, one could also do isolation on the process level. This originally started in 1979 with the inclusion of chroot (change root) in unix 7, giving the OS the ability to change the root directory for a given process and its children. This prevents the processes from accessing files that extend outside of the given root directory. In 2000, FreeBSD introduced an improved version of chroot, named jails, that aside from filesystem level isolation also provided additional sandboxing features, allowing it to isolate processes, networks, and users. Solaris improved upon jails creating a system called zones, a capability based isolation system, and building upon that to create Solaris containers. This prompted engineers at Google to develop their own version called process containers, which was later renamed to cgroups to prevent confusion with solaris containers. Cgroups were eventually merged into Linux 2.6.24, leading to the creation of Linux Containers, LXC.

In 2013, Solomon Hykes released Docker into open-source. Docker was designed to be a platform that abstracted away the intrecacies with using the lower level LXC apis and packaging. With the immense rise in popularity of Docker, the project was donated to the newly founded Open Container Initiative (OCI), which has now standardised APIs and packaging specifications across all container platforms.

Note

I really don't like this section, its just a lot of fluff.

Note

Im also only talking about 'modern' containers here

Unlike full virtual machines that exposes bare metal to the guest OS, containers reuse the kernel provided by the host OS. They are exposed to a small slice of the kernel's resources through the use of cgroups and namespaces. As a consequence of not needing to emulate a full operating system, containers are generally much more lightweight than virtual machines, allowing one to run hundreds of containers on a host, and reboot far faster as well.

todo

source

Most isolation features used for containers on linux are based on namespaces. For process isolation containers use both PID and IPC namespaces. PID namespaces provide a nestable way to isolate PID numbering, allowing each container their own personal PID space, including the ability to have their own init process, assisting in the shutdown and cleanup of containers. For communciation, IPC namespaces are used to prevent processes from communicating with processes outside of the container.

Moving on to filesystems, containers are provided their own filesystem, isolated using mount namespaces. The only exception to this is that kernel filesystems, such as sysfs and procfs cannot be namespaced, and have to be mounted directly for the container to function. This sadly allows the container to view the world outside of its namespace, but access to these filesystems is limited limited by docker, preventing them from writing or remounting kernel filesystems.

Network isolation is once again provided by the similarly named network namespaces, giving each container its own independent network stack including an individual IP address, routing tables, and network devices.

Sometimes containers need access to devices available to the system, such as graphics cards or peripherals. These can be granted to containers by virtue of a whitelist implemented using cgroups as they cannot be isolated using mount namespaces due to it being a virtual filesystem. The whitelist also disallows containers to create their own devices. Besides this, a container can also be created as a so called 'privileged' container, a container that has full access to all devices on the host machine.

Lastly, resources for containers are constrained using cgroups, allowing the host to place limits on CPU usage, memory usage, disk I/O, etc. Further enhancements to containers can be provided through the use of Linux Capabilities (much of which are disabled by default in Docker), SELinux, and AppArmor.

References

[1] D. Levinthal, "Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors," p. 72, 2008.

[2] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing OS abstraction," in Proceedings of the Fourteenth EuroSys Conference 2019, pp. 1–17, 2019.

[3] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in S&P, May 2019.

[4] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in USENIX Security, Aug. 2018.

[5] F. J. Corbató and V. A. Vyssotsky, "Introduction and Overview of the Multics System," in Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I, AFIPS '65 (Fall, Part I), (New York, NY, USA), pp. 185–196, ACM, 1965.

[6] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," Proceedings of the IEEE, vol. 63, pp. 1278–1308, Sept. 1975.

[7] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics," Commun. ACM, vol. 17, pp. 388–402, July 1974.

[8] E. L. Glaser, J. F. Couleur, and G. A. Oliver, "System design of a computer for time sharing applications," in Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I, pp. 197–202, ACM, 1965.

[9] J. H. Saltzer and F. Kaashoek, Principles of Computer System Design: An Introduction. Burlington, MA: Morgan Kaufmann, 2009.

[10] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in 2012 IEEE Symposium on Security and Privacy, pp. 523–537, IEEE, 2012.

[11] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay, "Measuring Real-World Accuracies and Biases in Modeling Password Guessability," in 24th {USENIX} Security Symposium ({USENIX} Security 15), pp. 463–481, 2015.

[12] T. Seitz, M. Hartmann, J. Pfab, and S. Souque, "Do Differences in Password Policies Prevent Password Reuse?," in Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems, CHI EA '17, (New York, NY, USA), pp. 2056–2063, ACM, 2017.

[13] E. Stobert and R. Biddle, "The Password Life Cycle: User Behaviour in Managing Passwords," in 10th Symposium On Usable Privacy and Security ({SOUPS} 2014), pp. 243–255, 2014.

[14] Trusted Computer System Evaluation Criteria ("Orange Book"). Department of Defense Standard 5200.28-STD, 1985.

[15] D. E. Denning, "A Lattice Model of Secure Information Flow," Commun. ACM, vol. 19, pp. 236–243, May 1976.

[16] R. Sandhu, "Lattice-based access control models," Computer, vol. 26, pp. 9–19, Nov. 1993.

[17] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn, "Role-Based Access Control (RBAC): Features and Motivations | NIST," in 11th Annual Computer Security Applications Conference, Dec. 1995.

[18] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-based Access Control," ACM Trans. Inf. Syst. Secur., vol. 4, pp. 224–274, Aug. 2001.

[19] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," Computer, vol. 29, pp. 38–47, Feb. 1996.

[20] S. Osborn, R. Sandhu, and Q. Munawer, "Configuring Role-based Access Control to Enforce Mandatory and Discretionary Access Control Policies," ACM Trans. Inf. Syst. Secur., vol. 3, pp. 85–106, May 2000.

[21] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," tech. rep., MITRE CORP BEDFORD MA, 1973.

[22] D. E. Bell, "Secure computer systems: A refinement of the mathematical model," tech. rep., MITRE CORP MCLEAN VA, 1974.

[23] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and multics interpretation," tech. rep., MITRE CORP BEDFORD MA, 1976.

[24] K. J. Biba, "Integrity considerations for secure computer systems," tech. rep., MITRE CORP BEDFORD MA, Apr. 1977.

[25] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," Commun. ACM, vol. 9, pp. 143–155, Mar. 1966.

[26] R. S. Fabry, "Capability-based Addressing," Commun. ACM, vol. 17, pp. 403–412, July 1974.

[27] P. J. Denning, "Before memory was virtual," In the Beginning: Personal Recollections of Software Pioneers, 1996.

[28] W. Lonergan and P. King, "Design of the B5000 system," Datamation, vol. 7, no. 5, pp. 28–32, 1961.

[29] A. J. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?," ACM SIGARCH Computer Architecture News, vol. 10, no. 4, pp. 3–10, 1982.

[30] T. Kilburn, D. B. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," IRE Transactions on Electronic Computers, no. 2, pp. 223–235, 1962.

[31] T. de Raadt, "OpenBSD 3.3." https://www.openbsd.org/33.html, May 2003.

[32] M. D. Schroeder, D. D. Clark, and J. H. Saltzer, "The Multics Kernel Design Project," in Proceedings of the Sixth ACM Symposium on Operating Systems Principles, SOSP '77, (New York, NY, USA), pp. 43–56, ACM, 1977.

[33] S. Brookes and S. Taylor, "Containing a Confused Deputy on x86: A Survey of Privilege Escalation Mitigation Techniques," International Journal of Advanced Computer Science and Applications, 2016.

[34] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Lightweight kernel isolation with virtualization and VM functions," in Proceedings of the 16th

ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 157–171, 2020.

[35] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 368–379, ACM, 2016.

[36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the Effectiveness of Address-space Randomization,” in Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS ’04, (New York, NY, USA), pp. 298–307, ACM, 2004.

[37] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, “Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software,” in 2006 22nd Annual Computer Security Applications Conference (ACSAC’06), pp. 339–348, Dec. 2006.

[38] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” in Presented as Part of the 21st USENIX Security Symposium (USENIX Security 12), pp. 475–490, 2012.

[39] T. Linux Kernel Development Community, “Ext4 Data Structures and Algorithms — The Linux Kernel documentation.” https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html.

[40] F. Cohen, “A cryptographic checksum for integrity protection,” Computers & Security, vol. 6, pp. 505–510, Dec. 1987.

[41] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “End-to-end Data Integrity for File Systems: A ZFS Case Study.,” in FAST, pp. 29–42, 2010.

[42] M. A. Halcrow, “Demands, solutions, and improvements for Linux filesystem security,” in Proceedings of the 2004 Linux Symposium, vol. 1, pp. 269–286, 2004.

[43] R. P. Goldberg, “Architectural principles for virtual computer systems,” tech. rep., HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS, 1973.

[44] G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures,” Commun. ACM, vol. 17, pp. 412–421, July 1974.

[45] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, “Intel virtualization technology,” Computer, vol. 38, no. 5, pp. 48–56, 2005.