OS Hardening

Making systems more secure

Seminar paper

Ausgewählte Kapitel der IT-Security

Author:
John Ostrowski

Student identification number:
1710475050

Date:
28.1.2020

Contents

Chapter 1

Introduction

The operating system (OS) serves as the intermediary between the computer's hardware and the application programs. It assists the user in performing operations on the underlying physical components and manages the interactions between different devices connected to a computer system. The two main activities are processing and the coordination of the input/output (I/O) devices connected to the computer system. Processing includes the management and scheduling of different user programs running simultaneously as well as the definition, management and preservation of data in memory.

Since multiple internal and external users and programs can interact with one computer system at the same time, the OS has to ensure that they only can read and write to files and devices where they have permission to. Otherwise, data can be stolen and used maliciously. [SGG13]

To reduce the attack surface of an operating system different procedures and policies need to be enforced to allow for a secure operation of a system. This process is called "operating system hardening"[PP09]

In this paper the term security is explored and applied to the security requirements of an operating system. First, the various threats are examined and subsequently techniques securing against them will be showcased. The focus of this paper is on creating safe execution environments for exploitable applications with buffer overflows. Additionally, techniques on containing the damage will be highlighted.

Chapter 2

Security Requirements

To understand the security requirements of an operating system one first has to understand how security is defined. There are many definitions for ,,security“ that focus on different use cases and deal with various aspects. Over the years the so-called ,,CIA Triad“ became established as the foundation blocks for defining a secure system.

2.1 Security Requirements

The CIA Triangle stands for Confidentiality, Integrity and Availability. These three properties must be ensured for any secure system. The following maxims are stated using the definition of [CJT12, Com91].

Confidentiality is the ability to allow only an authorized person to access protected information and keep sensitive information from prying eyes.

Integrity assures that information was not changed by recognizing that the data was modified.

Availability is the portion of time where a system provides access for its legitimate users.

An example of confidentiality is an access control list that defines who can read a specific file. In many UNIX-like operating system this is handled by POSIX permission model and controlled by the kernel [Gru03].

Integrity is a bit more complicated to ensure. Pfleeger et al. [PPM15] referred to the many definitions of integrity. Integrity can not only mean the realization that a document was changed. An example of this being a transmission error or an attack of a malicious actor, which can be noticed through a checksum. Integrity can also be voided on the logical level where data can get changed by an authorized user leading to a *logical integrity* error. An authorized user may change the state of information in a system to some logical incorrect presentation, like setting the age of a person to -3 years. In the above-mentioned definition of integrity, this would still be seen as preserved secrecy since the data was willingly modified. But as we know this is not possible and the logical integrity is voided.

Logical integrity is hard to protect, as the system cannot notice such error on its own. It needs the help of humans telling the system what states are allowed and which are not. This is a huge research area of requirement engineering that acknowledges the impossibility of this goal [Poh94].

An example of availability is the uptime of a cloud service. Many high available (HA) systems promote and benchmark their availability using the ,,nines“ scale. The uptime is usually given in percent and the more '9' they have the closer they are to a 100%. ,,Three nines“ would resemble 99.9% whereas ,,Six nines“ would resemble 99.9999. This hints that no system can achieve a hundred percent availability [PH01].

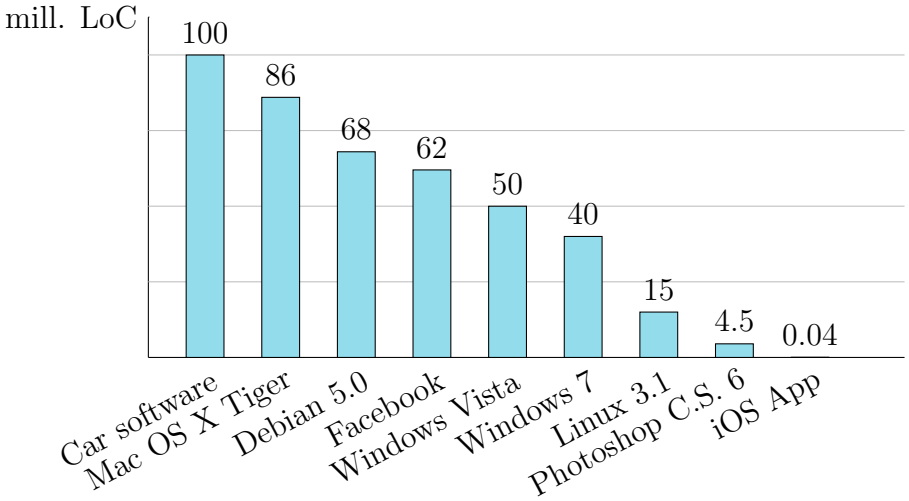
Other Criteria

Just ensuring the CIA Triad is not enough for many systems and require and depend on other important requirements. One being the *authentication* ensuring a person's identity. Usually, the identification of a person can be validated by something the person knows, like the answer to a certain question only he or she knows, or by something the person has, like a key to a house. Then there's the *non-repudiation* that refers to the characteristic of an author not being able to deny the authorship of a file. For example, a signature is not able to be disputed by the signer. There are many more which can be found in [Gra11].

2.2 Operating System Security Evaluation

An operating system needs to fulfill the criteria listed in 2.1 to be called secure. As already hinted, creating a completely secure system is hard if not impossible to achieve. Striving to such a goal gets even harder when dealing with enormous complex systems. Operating systems are one of the biggest code constructs (as seen in figure 2.2.1) in human history, proving a big attack surface for an adversary as there can be many software bugs in a million lines of code (mill. LoC).

Figure 2.2.1: Million Lines of Code [iB15]



TCSEC	ITSEC	CC	Description
A1	E6	EAL 7	formally verified design and tested
B3	E5	EAL 6	semiformally verified design and tested
B2	E4	EAL 5	semiformally designed and tested
B1	E3	EAL 4	methodically designed, tested, and reviewed
C2	E2	EAL 3	methodically tested and checked
C1	E1	EAL 2	structurally tested
D	E0	EAL 1	functionally tested

Table 2.1: Comparison between System Evaluation Criteria

There are three main evaluation criteria for testing and classify computer systems. The *Trusted Computer System Evaluation Criteria* (TCSEC) [US 85], developed under the United States Government Department of Defense and the *Information Technology Security Evaluation Criteria* (ITSEC) [Com91], published under the Commission of the European Communities [MB90] are standards of basic requirements for a computer system. These two standards got unified under ISO/IEC 15408 standard [ISO08b, ISO08a, ISO09] which is known under the name *Common Criteria*.

As depicted in figure 2.1 they classify computer systems on a set of criteria ranging from minimal protection (EAL 1) to verified protection (EAL 7). When applied to operating systems many fall under the category EAL 4, including Red Hat Enterprise Linux 7.1 [Red17], Windows XP [Sch05] and Windows 2008 [Sci09]. Apple Mac OS X 10.6 got rated for an EAL 3 [Bun10] and Ubuntu 16.04 for EAL 2 [Com19]. It has to be noted that examination takes a long time to conducted and only applies to one specific version. This is one of the reasons why such investigations are performed rarely. Additionally, they are very costly, leading to only a marginal gain for the product.

2.3 Common Threats

The main security purpose of an operating system is the separation of user data and applications. Other applications or users should not be able to access data they are not authorized to. The operating system has different techniques for achieving this goal. But before we look into the protection and hardening of an operating system, the most prominent attacks are presented. This list of threats is by no means complete and should only give a brief overview.

Application Exploit

Application exploits utilize programming errors to take advantage of the system. A good example is a malicious PDF that exploits a programming error in the PDF reader. PDFs are designed as a read-only document and therefore many users feel secure opening such a file, even if they received it from an untrusted source. This is not always true, as there can be mistakes made during the interpretation of PDF file by the PDF software. Adversary's can exploit such errors and execute arbitrary code, gaining access

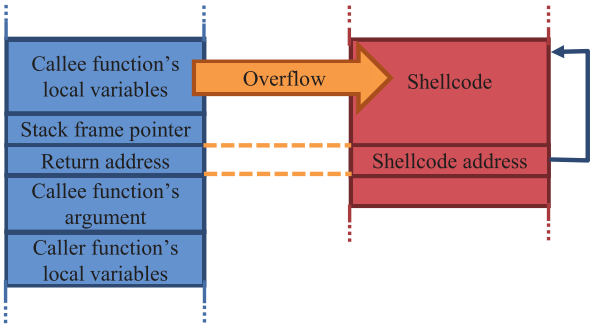


Figure 2.3.1: Buffer overflow [Oka15]

to the user's machine. Recently, researches found insecure signature and encryption handling of all of the 27 most used PDF readers that allowed modifying a PDF after it was signed and inserting arbitrary code into encrypted PDF files. For example, a payslip that is already signed or encrypted can be modified by adding ,,zeros"to the stated salary and thus increasing the displayed payout [MMM+19, MIM+19].

Buffer Overflow

Buffer overflows are a subtype of an application exploit. As they are one of the more common attacks out there we examine them closer. Buffer overflow is the practice of exploiting unprotected size-unrestrained memory buffers. The goal of such an attack is the modification of the return pointer of a program, enabling the attacker to jump to an arbitrary location in memory. This can be used to executing injected code. Such injected code can be *shellcode* allowing the attacker to access the user command terminal from the exploited process. This consequently gains the adversary the same privilege as the exploited program [Gra11].

There are different kind of buffer overflows like a *stack buffer overflow*, where the program stack is filled with exactly the correct amount of data to modify the return pointer, or a *heap buffer overflow*, which overwrites sections of the heap. The second method is harder to do, usually exploiting functions like *malloc*, whereas in the first method requires a size-unrestricted buffer which are used more often. This becomes increasingly more complicated with the prevention techniques showcased in chapter 3.

The function pointer is used to reference the memory address of the function (parent) that calls the currently executing function (child). As seen in figure 2.3.1, overwriting this return pointer allows to jump to any other part in memory. One could execute a function not normally accessible to the user or inserted code in memory. As mentioned before, shellcode can be written somewhere else in the program (eg. some other buffer) and be executed. As locating the inserted shellcode is not easy the attacker inserts a *NOP chain*, leading to the starting of the shellcode. NOP stands for the no-operation and as the name suggests the computer skips the operation till it reaches the starting point of some operation [Oka15, Gra11].

Other more advanced techniques include the use of return-oriented programming ROP, gadget chains and return-into-library methods. These techniques are used to overcome some prevention methods found in chapter 3.

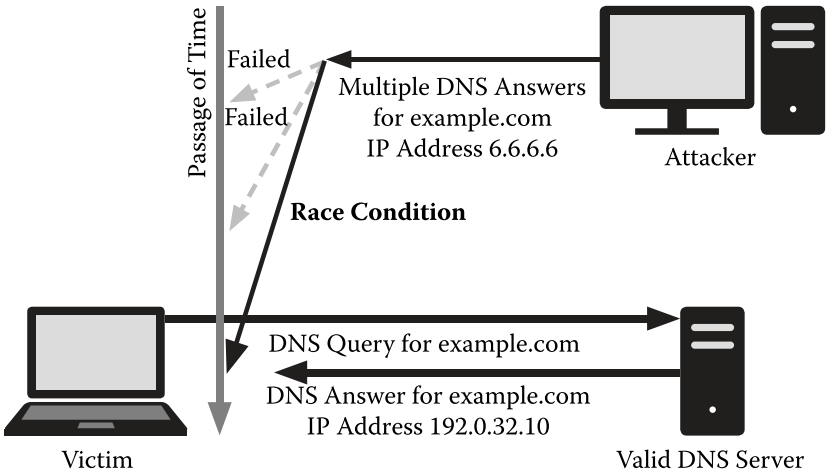


Figure 2.3.2: DNS request race condition [Gra11]

Race Condition

Race conditions can occur when concurrent processes can read or write on the same memory location. This attack is also known as *time-to-check and time-of-use* (TOC/-TOU) as a shared value is checked and subsequently used to exploit some kind of software bug. These software bugs occur because the programmer does not except that some other process can modify the shared value during executing. One of the better-known network TOC/TOU attacks is the injection of a fake DNS query as seen in figure 2.3.2. This attack exploits the DNS race condition, as any valid answer is seen as correct. There is a short time frame for an attacker to send a valid DNS response to the host before the server can answer. If the attacker can guess that the user sent out a DNS query for eg. example.com, the attacker can send back a DNS answer with a fake IP address before the DNS server can to it and redirect the host to the fake IP address.

TOC/TOU do not only apply to the network, but also to shared resources on the local machine. For example, an attacker can create a symbolic link to a file in the precise moment a privileged program wants to write to it. If this symbolic link points to */etc/passwd*, the privilege program unwillingly overwrites the password file, allowing the attacker root access to the machine [Gra11].

Other Attacks

There are many more techniques attacking the CIA requirements of an operating system. Many malicious actors attack the confidentiality as it usually has the most value to an adversary. The attacker can use this information to blackmail the victim and demand a ransom or use the information to their advantage. In the 2019 Verizon data breach investigations report [Ver19] as depicted in figure 2.3.3 the motives of threat actors were 67% financially motivated, 20% espionage driven, 8% for fun and self-esteem related.

Breaching the confidentiality of a system involves some kind of *privilege escalation* technique, exploiting a programming error to obtain a higher privilege level.

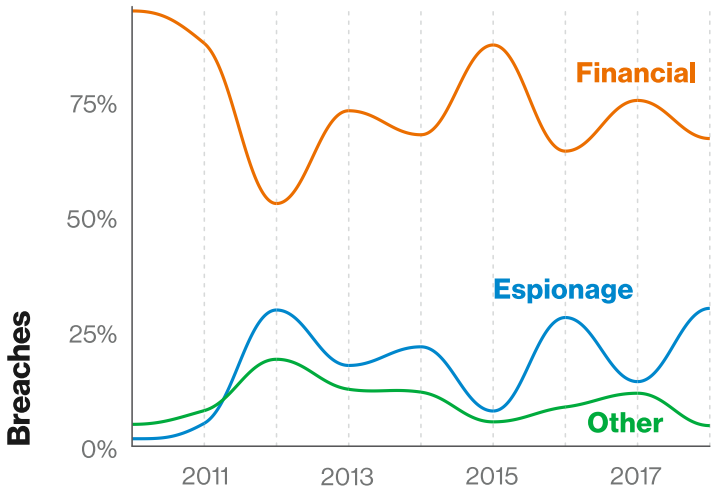


Figure 2.3.3: Threat actor motives in breaches over time [Ver19]

Other attacks on the CIA criteria include *SQL injections*, *web exploits*, *password cracking*, *denial-of-service attacks*, *eavesdropping attacks* and *malware*. This is just a short overview of the most common attacks on operating systems and will therefore not be discussed in detail.

Chapter 3

OS Hardening

It is to be noted that achieving a completely secure system is almost impossible. What makes this especially challenging is the fact, that a general-purpose operating system has to account for a diverse set of applications running on it. The operating system has only a small influence changing the behavior of a program, as the source code is mostly controlled by a third party. It is the job of the operating system to manage all of the programs and create a safe environment for the user and the applications.

Since the operating system can not change the way a program works internally it has to use other methods to protect against various threats. The main goal of *OS hardening* is to *reduce the attack surface* by creating *safe environments* for applications, *remove unnecessary services* and introduce a *resource control* with limited access [SGG13].

In this chapter various methods on creating safe environments are shown on the example of a stack buffer overflow. Next, we will dive into securing the control of files and applications. At the end, there is a short section on removing unnecessary services and reducing the overall attack surface of an operating system.

3.1 Safe Environments

Creating a safe environment for applications is not an easy process at all. The operating system has to accommodate many different applications that operate on the same shared hardware. The OS administrates the execution of programs and is in charge of the separation of concerns.

Even though all applications operate in the same memory and on the same CPU, they should not be able to access files that they are not authorized to. Creating such low level executing environments on the hardware level is very complex and not part of this paper, although also belonging to the security of an operating system.

In this paper we will not concentrate on the creation of executing environments but more on the securing of such. The line between creating and hardening is hard to tell, as they go hand in hand.

This section will investigate a buffer overflow attack and how an operating system can prevent the exploitation of a programming error.

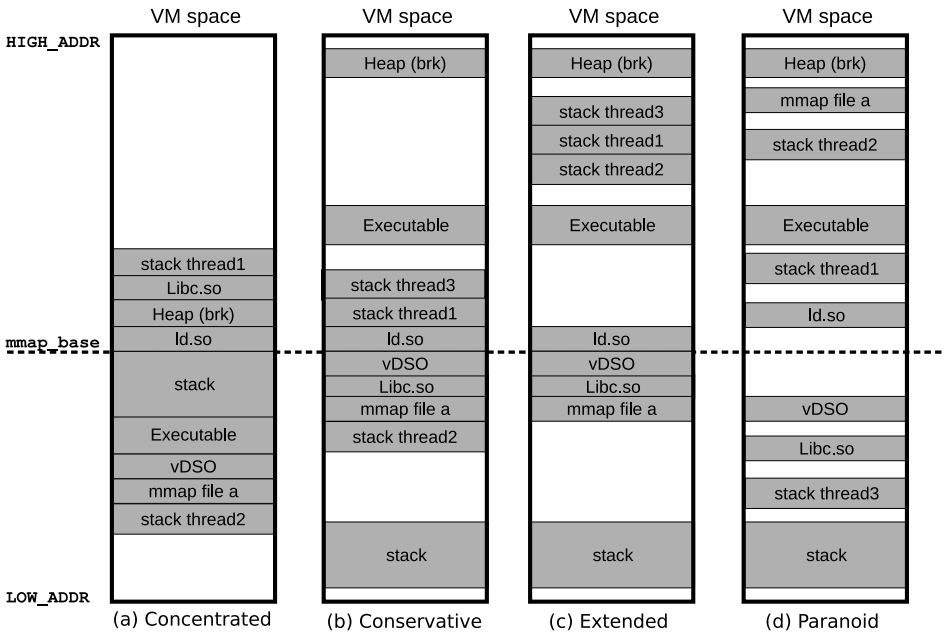


Figure 3.1.1: Address Space Layout Randomization by ASLR-NG [MRR19]

Protection against a Buffer Overflow

As hinted by OWASP [OWA14] buffer overflows are almost exclusively present in strongly-typed programming languages that allow direct memory access like C/C++ and Assembly. For other languages that do not have direct memory access like Java, Python and .NET hardly have the problem of a buffer overflow as the OS and additionally, the compiler or the executing environment make sure that the program does not have any access to unauthorized memory.

Banning programming languages that have direct memory access is nonsensical as there must always be a way of interacting with a low-level interface. Therefore, other mechanisms must be in place to prevent the overwriting of the return pointer, thus prohibit the execution of arbitrary code.

There are two main different protection techniques; one that is enforced by the kernel and one that is enforced by the compiler [SJ04].

- **Kernel-Enforced Protection**: The kernel does not know the internal functionality of the executing program, therefore it can only modify the layout of the memory and enforce access control rights.
  - **Memory Access Control**: By creating non-executable (NOEXEC) memory spaces. This prevents the execution of injected shellcode in the heap or the stack of an application.
  - **Memory Address Randomization**: Address Space Layout Randomization (ASLR) introduces randomness of the placement in virtual memory (see fig. 3.1.1). For an attacker it is therefore hard to know where the location of variables, binary, libraries, heap and stack are in memory, as it is different for each execution. Pointing to injected shellcode is very hard since the attacker does not know where the OS placed it.

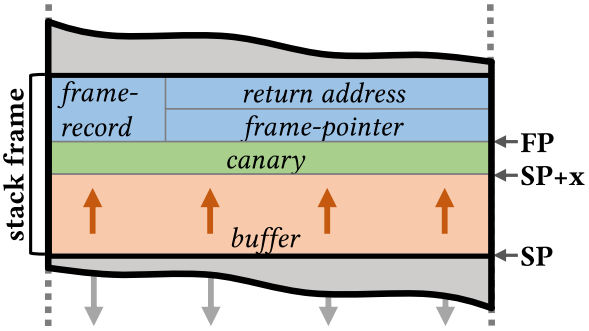


Figure 3.1.2: Compiler-Enforced Protection: Stack Canaries [LGN+19]

- **Compiler-Enforced Protection**: This method tackles the problem during the compilation of the program. The compiler leverages the knowledge it has on the structure of the program and could modify it in the way of securing against buffer overflows.
  - **Stack Canaries**: The compiler can insert special data, called *canaries*, into different parts of the program memory that get checked during the program execution. If an attacker tries to overflow a buffer, the possibility of overwriting a canary is very high, since they are placed in strategical positions as seen in figure 3.1.2. If the value of a canary changes during the execution of a program the system will notice and halt the application.

There are different implementations of buffer overflow prevention techniques. In Linux these systems are mostly maintained by the PaX team, joined by Exec Shield, kNoX, RSX, OpenWall Project and Immunix. There are multiple executable space protections such as PAGEEXEC, MPROTECT. For ASLR implementation there are RANDUSTACK and RANDEXEC. For compiler-based protection there is the well-known implementation StackGuard and StackShield. On the Windows OS they provide security through NGSEC's StackDefender [SJ04].

Virtualization

There are several security benefits of executing applications in a virtualized environment that encapsulat it from the surroundings. It is a more extreme version of the aforementioned buffer overflow prevention techniques, since the program can only access information in the virtualized container, which provides restricted disk, network and input/output (I/O) operations. This is also known as *sandboxing* because, just like a kid, the program is only allowed to use the tools that are provided in the sandbox and cannot escape. It is to be noted that sandboxes and virtualization environments are not perfect and thus there are ways of escaping them.

3.2 Access Control

Access control on an operating system is needed to preserve the confidentiality of a multi-user and multi-application system. The system has to control who is allowed to access, modify and execute certain data. This is achieved by permission bits.

In Linux basic file access control is based on the traditional UNIX file model [KGB10]. Each file and directory has a total of 9 bits and some special markers to set the permission. Of those 9 bits three are assigned to the owner, another three are assigned to the owner's group and the rest belongs to all other users on the system. The three bits resemble the privilege to read (r), write (w) and execute (x) a given file.

In listing 3.1 the permission for the file */etc/passwd* is shown using the command *ls -l*. The command shows that this file is owned by the root user and the root group. To the left, the permissions are defined. The first dash (–) represents the type of the file. The next nine bits show the corresponding permission explained before. In this case, the owner is allowed to read and write the file, the group and all others are allowed to read it. Underneath, the *stat* list more detailed information about the file. It also displays the identification number of the owner (Uid) and the group identification number (Gid).

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root root 2522 Dec 19 10:31 /etc/passwd

$ stat /etc/passwd
File: /etc/passwd
Size: 2522        Blocks: 8        IO Block: 4096 regular file
Device: 10301h/66305d Inode: 6820215   Links: 1
Access: (0644/-rw-r--r--) Uid: ( 0/ root) Gid: ( 0/  root)
Access: 2020-01-08 12:27:05.299964716 +0100
Modify: 2019-12-19 10:31:46.720795305 +0100
Change: 2019-12-19 10:31:46.720795305 +0100
Birth: -
```

Listing 3.1: Print Access Permissions on Linux

Access Control List

Dividing the access control into three groups can be quite limiting, especially if there are users that need special permissions. Maybe we want to grant permissions to additional users or groups. This is where the traditional system becomes quite constricting. Therefore, an extension like access control lists (ACL) exists to provide a more flexible mechanism to manage permissions. The ACL integrated into Linux is developed using the POSIX.1 standard and combines with the traditional UNIX permission mode nicely [KGB10].

With the extended POSIX.1 ACL each file system object is associated with a set of access control entries (ACE) that combine the UNIX-style mode with two additional attributes for the user and the group. This allows for creating exceptions, creating a finer separation of privileges.

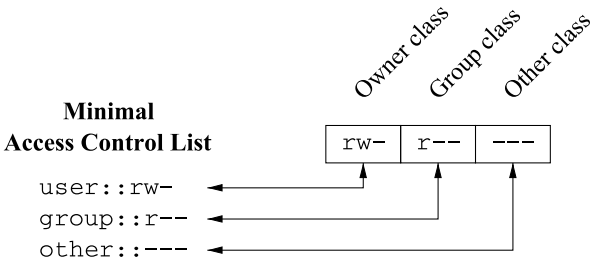


Figure 3.2.1: Simple ACL [Gru03]

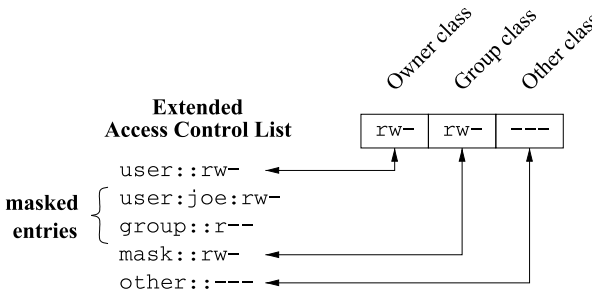


Figure 3.2.2: Extended ACL [Gru03]

Depicted in figure 3.2.1 we can observe the restrictiveness of a simple ACL allowing only three base classes. In figure 3.2.2 one can create additional definitions making it an extended ACL. An extended ACL provides the definition of additional user (*named user*) and additional groups (*named group*). In the example given, the named user Joe is granted read and write permission no matter in which group he belonged before. If needed, additional named groups can be defined.

Note that there is another entry called *mask*. This represents the highest privilege of all additional added entries and the traditional owning group permissions. For this example, the owning group has fewer permissions than the user Joe. Like an operational OR the permissions are added together and represented by the mask (group:r OR joe:rw = rw). The mask can be adjusted, making it possible to downgrade the permissions of all additional ACL entries. Setting the mask to read and execution access (r - e), would change the permission of Joe to read (r - -), since the mask defines the maximal possible permission set. The extended POSIX ACL is integrated into Linux since 2002 [Gru03]. With the command *getfacl* one can view the extended entries and with the command *setfacl* one can modify the permissions, as seen in listing 3.2

```
$ getfacl example.txt                                    // view extended ACL
# file: example.txt
# owner: root
# group: root
user::rw-
group::r--
other::r--

$ sudo setfacl -m "u:someuser:rwx" example.txt           // modify or add permissions
$ sudo setfacl -m "g:somegroup:r-x" example.txt
$ getfacl example.txt
...
user:someuser:rwx
group:somegroup:r-x

$ sudo setfacl -x user:someuser example.txt              // remove permissions
$ getfacl example.txt
# file: example.txt
# owner: root
# group: root
user::rw-
group::r--
group:somegroup:r-x
mask::r-x
other::r--
```

Listing 3.2: Extended ACL

Manditory Access Control

The aforementioned method of access control is called the *Discretionary Access Control* (DAC) method and is the most common mechanism to enforce confidentiality. The idea behind DAC is that the owner specifies who can access an object. This decision is based on the owner's discretion.

The huge vulnerability of this model lies in the fact that the objects run on the behalf of an authorized user and therefore has the same privileges. Any flawed application (eg. a possible buffer overflow) can now be exploited and the modified process can perform actions in the name of the owner.

The so-called *Mandatory Access Control* (MAC) tries to tackle the problem with a changed fundamental concept. Instead of letting the user decide the permission of objects, each user is given a certain clearance and each object is given a security classification. Only when a user has a higher clearance level as the required security classification (secret, top-secret, confidential) the system allows access.

In MAC-based systems the classification is outsourced from the owner to the system. The system restricts access by labeling security-related information to users and objects. Therefore, access rights of an object are decoupled from the user [FSC08, NC14].

Role-Based Access Control

Role-Based Access Control (RBAC) is independent of the policy model and can be used for either DAC or MAC systems. Here, a set of roles are assigned to users (eg. Human Resource, Research & Development, ...) and fitted with a set of rules and privileges. This makes RBAC systems especially useful in large scale environments [FSC08].

SELinux and AppArmor

Security Enhanced Linux (SELinux) was a research project by the NSA allowing for a MAC-based model on Linux OS and includes various rule definitions as well as RBAC. SELinux was made open-source by the NSA attracting many volunteers in the open-source community. The Red Hat team offered support for SELinux in their OS and contributing considerably to the project. Since 2003 SELinux is merged into the mainline of the Linux kernel [FSC08].

AppArmor developed by Immunix is another popular MAC-based extension for Linux OS. Compared to SELinux label based security, AppArmor is path-oriented. This means that instead of classifying every object with a label, AppArmor assigned access rights based on the pathname an object is placed in [Cow15].

This makes AppArmor inherently more simple, but less customizable than SELinux. AppArmor is installed into popular Linux distributions Ubuntu, Mint and Debian.

3.3 Reducing the Attack Surface

Reducing the attack surface is a vital part of securing the operating system. As seen in section 3.2 we can use access control to prevent flawed applications to breach confidentiality. We should not rely on this technique entirely and should reduce the possible attacks in the first place.

There are several procedures for achieving this goal. These techniques are highlighted briefly, as they are not the main focus of this paper. The following listing should give a glimpse into other OS hardening techniques and rounds of the overall view on this topic. The following recommendations are based on [Gra11, SGG13].

Removing unnecessary services to reduce the chance of an exploitable application.

Secure the network by using *Firewalls*, setting up rules for incoming and outgoing traffic and only allowing encrypted communication.

Securing the hardware by physically ensuring that no malicious actor can access the computer system. Locking down I/O like USB ports reduce the attack surface that is hard for an operating system to secure against.

Secure passwords are needed to ensure the authorization and thus the confidentiality of a system.

Chapter 4

Conclusion

For an operating system to be called secure it needs to ensure the three criteria of the CIA Triad, namely confidentiality, integrity and availability. Securing all three maxims perfectly is hard to do, especially as the applications running on the OS cannot be trusted. As no system is perfect the operating system has to compensate this through different OS hardening techniques. The various techniques can be grouped into three main methods; creating safe environments for the application is an essential part of keeping itself from other applications protected and vice versa. Creating safe executing environments can be achieved through the use of NOEXEC memory, ASLR and stack canaries. Another way of hardening the OS is by enforcing access control measures, like MAC, DAC and RBAC. Creating a strict access control mechanism protects the confidentiality of data and access to applications and therefore improves the overall security of an operating system.

Bibliography

[Bun10] Bundesamt für Sicherheit in der Informationstechnik. BSI-DSZ-CC-0536-2010 for Apple Mac OS X 10.6 from Apple Inc., 2010. 4

[CJT12] Saman Shojae Chaeikar, Mohammadreza Jafari, and Hamed Taherdoost. Definitions and Criteria of CIA Security Triangle in Electronic Voting System. *International Journal of Advanced Computer Science and Information Technology*, 2012. 2

[Com91] Commission of the European Communities. *Information Technology Security Evaluation Criteria (ITSEC): Provisional Harmonised Criteria*. Office For Official Publications Of The European Communities, Luxembourg, 1991. 2, 4

[Com19] Common Criteria. Certified Products: New CC Portal. `https://www.commoncriteriaportal.org/products/`, 2019. (Accessed 6 Jan. 2020). 4

[Cow15] Crispin Cowan. Securing Linux Systems with AppArmor. *Novell*, 2015. 13

[FSC08] Luis Franco, Tony Sahama, and Peter Croll. Security Enhanced Linux to Enforce Mandatory Access Control in Health Information Systems. 2008. 13

[Gra11] James Graham. Cyber Security Essentials. *Auerbach Publications*, 2011. 3, 5, 6, 14

[Gru03] Andreas Grunbacher. POSIX Access Control Lists on Linux. *USENIX Annual Technical Conference, FREENIX Track*, 2003. 2, 12

[iB15] Information is Beautiful. Million Lines of Code. `https://informationisbeautiful.net/visualizations/million-lines-of-code/`, 2015. 3

[ISO08a] ISO/IEC 15408-2:2008. Informational technology – Security techniques – Evaluation criteria for IT security, Part 2: Security functional components (2008), 2008. 4

[ISO08b] ISO/IEC 15408-3:2008. Informational technology – Security techniques – Evaluation criteria for IT security, Part 3: Security assurance requirement (2008), 2008. 4

[ISO09] ISO/IEC 15408-1:2009. Informational technology – Security techniques – Evaluation criteria for IT security, Part 1: Introduction and general model (2009), 2009. 4

[KGB10] Aneesh Kumar, Andreas Grünbacher, and Greg Banks. Implementing an advanced access control model on Linux. 2010. 11

[LGN+19] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Protecting the stack with PACed canaries. *SysTEX '19, Association for Computing Machinery*, Article 4, September 2019. `http://arxiv.org/abs/1909.05747`. 10

[MB90] E. Mate Bacic. The Canadian trusted computer product evaluation criteria. In *[1990] Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 188–196, Tucson, AZ, USA, 1990. IEEE Comput. Soc. Press. `http://ieeexplore.ieee.org/document/143768/`. 4

[MIM+19] Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk. Practical Decryption exFiltration: Breaking PDF Encryption. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS '19*, pages 15–29, London, United Kingdom, 2019. ACM Press. `http://dl.acm.org/citation.cfm?doid=3319535.3354214`. 5

[MMM+19] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 1 Trillion Dollar Refund: How To Spoof PDF Signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS '19*, pages 1–14, London, United Kingdom, 2019. ACM Press. `http://dl.acm.org/citation.cfm?doid=3319535.3339812`. 5

[MRR19] Hector Marco-Gisbert and Ismael Ripoll Ripoll. Address Space Layout Randomization Next Generation. *Applied Sciences*, 9(14), July 2019. `https://www.mdpi.com/2076-3417/9/14/2928`. 9

[NC14] Vasudevan Nagendra and Yaohui Chen. Access Control Lists in Linux & Windows. *Stony Brook University*, 2014. 13

[Oka15] Takeshi Okamoto. SecondDEP: Resilient Computing that Prevents Shellcode Execution in Cyber-Attacks. *Procedia Computer Science*, 60:691–699, 2015. `https://linkinghub.elsevier.com/retrieve/pii/S1877050915023388`. 5

[OWA14] OWASP (Open Web Application Security Project). Buffer Overflows - OWASP. `https://www.owasp.org/index.php/Buffer_Overflows`, September 2014. (Accessed 8 Jan. 2020). 9

[PH01] Floyd Piedad and Michael Hawkins. *High Availability: Design, Techniques, and Processes*. Enterprise Computing Series. Prentice Hall PTR, Upper Saddle River, N.J, 2001. 3

[Poh94] Klaus Pohl. The three dimensions of requirements engineering: A framework and its applications. *Information Systems*, 19(3):243–258, April 1994. https://linkinghub.elsevier.com/retrieve/pii/0306437994900442. 3

[PP09] P K Patra and P L Pradhan. Hardening of UNIX Operating System. *Int J. of Computer Communication and Technology*, 1, 2009. 1

[PPM15] Charles P. Pfleeger, Shari Lawrence Pfleeger, and Jonathan Margulies. *Security in Computing*. Prentice Hall, Upper Saddle River, NJ, fifth edition edition, 2015. 2

[Red17] Red Hat. Red Hat Adds Common Criteria Security Certification for Red Hat Enterprise Linux. https://www.redhat.com/en/about/press-releases/red-hat-adds-common-criteria-security-certification-red-hat-enterprise-linux, 2017. (Accessed 6 Jan. 2020). 4

[Sch05] Bruce Schneier. Microsoft Windows Receives EAL 4+ Certification - Schneier on Security. https://www.schneier.com/blog/archives/2005/12/microsoft_windo.html, 2005. (Accessed 6 Jan. 2020). 4

[Sci09] Science Applications International Corporation Common Criteria Testing Laboratory. Microsoft Windows Vista and Windows Server 2008 Security Target. https://www.niap-ccevs.org/Product/Archived.cfm?par303=Microsoft%20Corporation, 2009. 4

[SGG13] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating System Concepts*. 2013. OCLC: 891662608. 1, 8, 14

[SJ04] Peter Silberman and Richard Johnson. A Comparison of Buffer Overflow Prevention Implementations and Weaknesses. *iDefense*, 2004. 9, 10

[US 85] US Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria. In US Department of Defense, editor, *The 'Orange Book' Series*, pages 1–129. Palgrave Macmillan UK, London, 1985. 4

[Ver19] Verizon. 2019 Data Breach Investigations Report. Technical report, 2019. 6, 7