ICT Risk Assessment - Appunti

Francesco Lorenzoni

September 2023

Contents

Chapter 1

Introduction

19 - Settembre

1.1 Info and Contact

Info on the exam...

Question time Monday from 16:00

1.2 ICT Security Fundamentals

A system security policy should preserve:

1. **Integrity** - Only users allowed to *update* certain information are actually able to update it
2. **Availability** - Users who want to *use* the system must be able to use it in a finite and reasonable amount of time
3. **Confidentiality** - Only users allowed to *read* certain information are actually able to read it

Depending on the adversary, the terminology changes

$$\text{Natural events} \Rightarrow \textit{Safety}$$

$$\text{Malicious and intelligent} \Rightarrow \textit{Security}$$

There are other properties regarding systems and their security:

1. **Robustness** evaluates how well the system resists and it is not violated by an attack
2. **Resilience** evaluates how well a system recovers and resumes its normal behaviour after it has been violated
   Resilience is preferred to robustness because it is more cost effective
3. **Vulnerabilities** are defects in the system which reduce robustness, hence safety and/or security. NOT every *defect* is a vulnerability, but every vulnerability is a *defect*.

Secondarily there are properties derived from *security*:

1. Traceability - Discover who has invoked a given operation
2. Accountability - Those who use a resource should pay for it
3. Auditability - Whether the security policy is enforced and satisfied
4. Forensics - Proving that some action has occurred and who has executed it
5. Privacy/GDPR - Who can read and update a certain information

Forensics is distinguished by *Traceability* because it is related to what can be proved in a court of law, not only who performed a single operation on the system. Forensics refers to a set of information able to convince a non-expert that something

21 - Settembre

1.3 Security policy - Glossary

First of all an **Asset analysis** is required. It is mandatory for a business to determine which resources are critical for their system to work, allowing to focus security efforts on specific assets.It is also crucial to determine the **impact**:

- ◇ A business process is stopeed (integrity or availability)
- ◇ A resource has to be rebuilt ex novo (integrity)
- ◇ Attacker discovers the information in the resource (confidentiality)

Asset *discovery* is usually done through an application installed in specific assets and discovers all the assets in the company network.

An **externality** is a cost or benefit incurred or received by a third party that has no control over the creation of that cost or benefit. It's important also to consider this because often the security of an ICT system may depend on third parties factors and entities, whose security isn't controlled by the owner of the system.

Security is a job *shared* by many individuals, hence there may be some **free riders**, i.e. individuals who tend to shirk and be negligent, and whose lack of effort affects security.There may be three prototypical case which define on which individuals depends the security of a system:

1. **Weakest-link**: security depends on agents with the *lowest* benefit-cost ratio. (worst-case scenario)
2. **Best shot**: security depends on agents with the *highest* benefit-cost ratio.
3. **Total effort**: security depends on the sum of efforts of **many** agents.

Chapter 2

Security Policy

21 - settembre

2.1 Definining Security Policy

A **Security Policy** is a set of rules that an organization adopts both to minimize cyber risk and to define the goals of security; it must:

- ◇ Define goals of security : assets and resources to protect assets
- ◇ Define the correct behaviour of all users
- ◇ Forbid dangerous behaviours and components
- ◇ Imply the definition of:
  - ...
- ◇ Avoid violating the legislation that concerns ICT systems

2.2 Terminology

2.2.1 Subject and Object

- ◇ **Subject**: entity which can invoke an operation on an object
  Subject or *Principal*
  ; e.g. User, application, program, process, ...
- ◇ **Object**: Instance of abstract data type; e.g. function, variable, logical or physical resources

An *object* which invokes operations on other object is both an object and a subject.

2.2.2 Access Rights

If subject S is entitled to invoke operation $\mathcal{A}$ on object Obj, then S owns an access right on $\mathcal{A}$ on Obj.
Access rights can be directly or indirectly deduced from the security policy and from the adopted implementation:

- ◇ **Direct**
  $$Scan\ read\ file F \Rightarrow S owns\ a\ read\ right\ on\ F$$
- ◇ **Indirect** Any program P - executed by S which reads the memory segment MS in which F is stored - owns a read right on MS

2.3 Composite policy

A whole security policy is the result of the composition of 9 more specific policies.

2.3.1 AUP - Acceptable Use Policy

2.3.2 ACP - Access Control Policy

2.3.3 Change Management Policy

Refers to formal process for making changes to the ICT system, including software and hardware updates, third parties dependencies...

2.3.4 Information Security Policy

Critical one, determines which users/applications can invoke object operations that reads and manipulates system information.

2.3.5 Incident Response Policy

2.3.6 Remote Access Policy

Defines acceptable methods for accessing remotely assets in the system internal network

2.3.7 Email/Communication Policy

Defines how employees can use business communication medias, and what contents they can share through them.

2.3.8 Disaster Recovery Policy

Defines how to behave if an event has a significant business impact

2.3.9 BCP - Business Continuity Plan

2.4 ISP - Information Security Policy

Determines which subject can invoke object operations that reads and manipulates system information.
Owner may choose to structure the policy in two ways:

- ◇ Default **allow**: policy defines *forbidden* operations
- ◇ Default **deny**: policy defines *legal* operations

Secondarily, one must decide the owner's degree of freedom:

- ◇ **Discretionary** Access Control: no constraints, the owner is free (commercial world)
- ◇ **Mandatory** Access Control: some contraints the owner cannot violate (military/defence world)

2.4.1 Six Dumbest Ideas in Computer Security

1. Default Allow
2. Enumerate Badness
3. Penetrate and PAtch
4. Hacking is Cool
5. Educating Users
6. Action is Better then Inaction

The first two points are strongly related. The reason for 1. is that dangerous behaviours, i.e. things to forbid, are much more than legitimate ones, so the choosing *Default allow* implies to enumerate badness (2.), i.e. things to be forbidden.

2.4.2 Discretionary Access Control

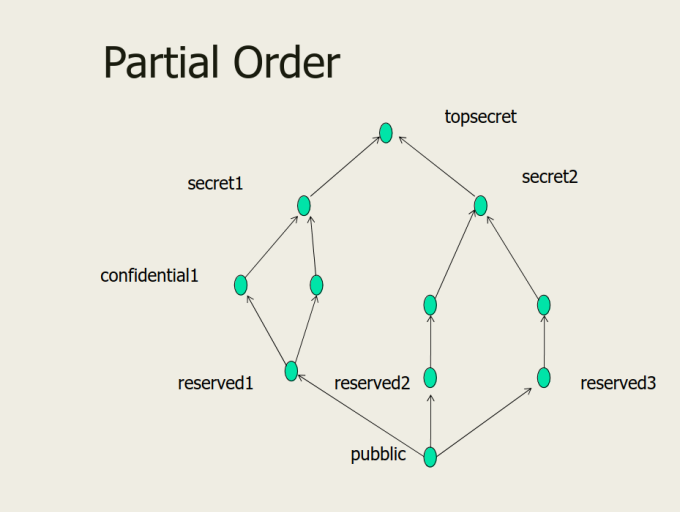
For each object there is an:

- ◇ owner of the ICT system
- ◇ of the business process that uses the object

The owner is unconstrained and decides the rights of all other users on its own objects.

2.4.3 Mandatory Access Control

All objects and subject must be partitioned into partially ordered[1] classes (possibly the same set of classes for obj and subj, it would simplify). S may be granted the right to invoke $\mathcal{A}$ on Ob only if the classes of S and the one of Obj satisfy a predefined condition that does not depend upon S or Obj.



22 - Settembre

2.5 MAC Policies

2.5.1 Bell-LaPadula

Any subject S in class C is **allowed** to:

- ◇ Read any file in a class $D \leq C$
- ◇ Write any file in C class
- ◇ Append to any file in a class $D \geq C$
- ◇ Grant rights if S is the owner and the previous contraints are satisfied

This a *no write-down* policy, which preserves confidentiality and prevents an information flow from a higher to a lower level, but may lead to information clustering in higher levels

2.5.2 Biba

- ◇ Read any file in $D \geq C$
- ◇ Write any file in $D \leq C$

Also called *no write-up* policy, guarantees integrity but sacrificing condidentiality.

2.5.3 Watermark

Time-dependant

2.5.4 No interference property

2.5.5 Clark-Wilson

2.5.6 Chinese Wall

As soon as S invokes and operation on $Obj \in C$, then:

- ◇ S cannot invoke operations on Objects $\notin C$
- ◇ S can invoke operations on Objects $\in C$

[1] Check image below for an example

2.5.7 Overall Policy

In a real world context, organizations merge several of the above-mentioned policies. For a subject, the may be two distinct levels, one for *confidentiality* and one for *integrity*

2.6 Trusted Computing Base

TCB includes any modules involved in the implementation of the security policy. TCB modules are critical, because any bug in one of these represents a vulnerability.
It is important to note that the security level (and the trust in it) is *inversely proportional* to the **size** of the TCB, since more lines of code imply higher chance to hide bugs, hence vulnerabilities.

2.7 Representing Security Policy

It is possible to build an Access Control Matrix, where `ACM[i,j]` indicates which operations subject `I` can invoke on object `j`.
Some kind of representation of ACM is a **necessary** and **not sufficient** condition to consider a policy valid and system actually secure.

		O	
S			
		Rights	

Table 2.1: Access Control Matrix

Chapter 3

Vulnerability, Attack, Intrusion

3.1 Vulnerability

A **vulnerability** is a defect in a person, a component or a set of rules which enables a threat agent to execute an **attack**: action that grants access rights that violate the security policy.

In short,

*A **vulnerability** is a bug that enables an **attack***

Every vulnerability is a bug, but *not* every bug is a vulnerability

3.2 Attack

An **attack** is an action and/or the execution of some code that may grant to the person or the module that executes it some illegal access rights, and it is related to a vulnerability that enables it.
The output of an attack is **stochastic**, it may fail according to a probability distribution.

3.3 Threat Agent

A **threat agent** is a source of **attacks**, it may be *natural* (floodings, earthquakes...) or *man-made* (adversary with a goal). *Man-made* may be malicious or random (employee which clicks on something dangerous accidentally).

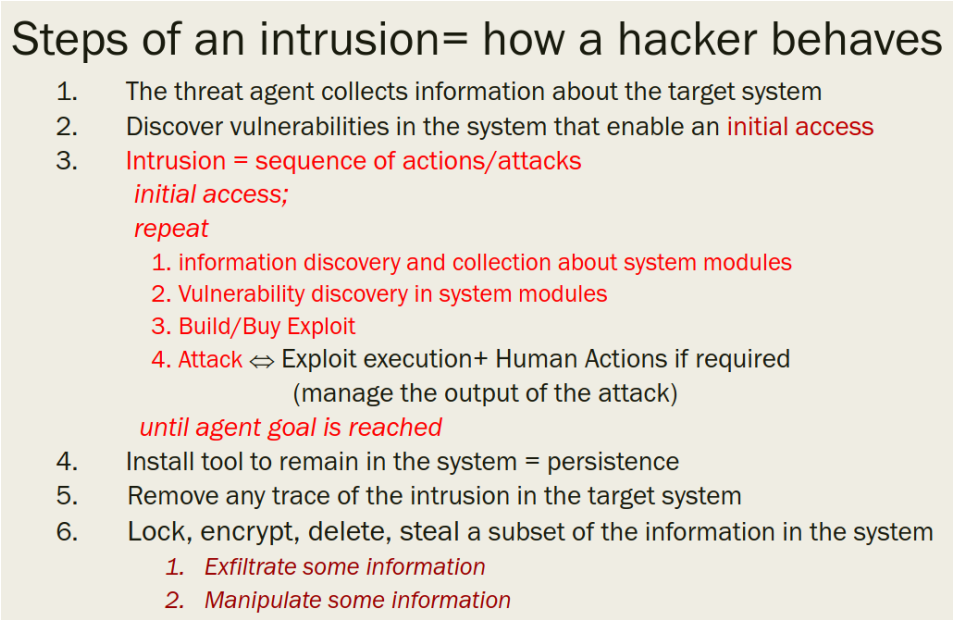
It is possible to **assess risk** only if assets, vulnerabilities and threat agents are known for a given system.

3.4 Intrusion

An **intrusion** is a sequence of *actions* and *attacks* of a threat agent to reach its goal, which initially owns its legal access rights and aims to gain illegal ones, hoping to control — a subset of — an ICT/OT system. Some actions may be actuals attacks, while others may collect information to discover possible attacks. Such actions (and attacks) can be implemented by a program called *exploit*.

Once a threat agent gained control over an ICT/OT system:

- ◇ Collect and exfiltrate infomration from the system
- ◇ Update any information in the system
- ◇ Prevent access to any resource/information in the system



The steps of an intrusion include a recursive phase highlighted in red in the picture; it appears clear that an attacker cannot plan an entire intrusion in advance before starting it, since an attack reveals information and (possibly) vulenon the system which the next attack will be based on.

3.5 Initial Access

A set of techniques that adversaries may use in an intrusion as entry vectors to gain an initial foothold within an ICT/OT environment.

Informations gathered through initial access are sold on the deep web to hackers team who aim to penetrate a system.

3.6 Countermeasure

The **attack chain** is the sequence of *useful* attacks in an intrusion. A defendant wants to increase the number of *useless* attacks to slow down an intrusion. Besides, it is not mandatory to remove *all* vulnerabilities to prevent an in intrusion, but even only one may be sufficient to interrupt te *attack chain*, thus preventing the attacker from collecting information that would lead to further attacks.

There are two main approaches when considering security:

- **Unconditional security**: Assume that any vulnerability will be exploited regardless of costs and complexity
- **Conditional security**: Consider who is interested in attacking the system and which vulnerabilities their intrusion can exploit.

3.7 Risk assessment

To wrap up, let's define what **Risk assessment and management** involves, keeping in mind that *cyber risk* resembles the average loss for instrusions.

1. Asset analysis
2. Threat agent analysis
3. Vulnerability analysis
4. Adversary emulation
5. Impact analysis
6. Risk evaluation and management: *compute and minimize loss*
   - Compute the risk
   - Accept some risk
   - Reduce some risk (countermeasures + scheduling)
   - Transfer residual risk (insurance)

Chapter 4

Vulnerabilities

In this chapter we'll take a deeper look into vulnerabilities and the related attacks, providing some examples and details.

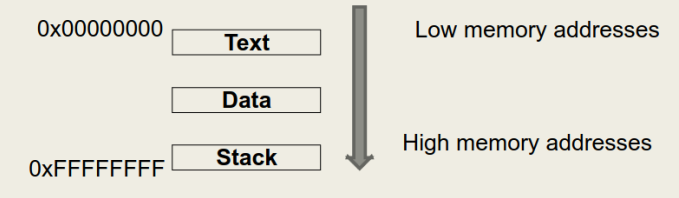
A dummy but instructive vulnerabilties classification distinguishes two major classes:

1. **Local vulnerability** → vulnerability in a single *module*. Even if the vulnerability needs other modules to be exploited, the defect it mostly depends on is in a single module, and such kind of vulnerabilities can be removed by updating the modules they're dependant on.
2. **Structural vulnerability** → a vulnerability which arises as a result of the composition of multiple modules. Also called *emerging* vulnerability.

4.1 Local vulnerabilities

A basic example of a local vulnerability, is **memory overflow**, particularly easy to exploit in software written in C, due to its memory management. Basically, it is possible to *inject code* in a small memory area by inserting more data than the actual available space. It is common for many attacks to inject *code* where the software expects *data*. Another known example is SQL injection.

It is possible to inject code exploiting **stack overflow**. stack memory areas start at the highest memory addresses and grow backwards, towards lower addresses.



Thus it is possible to inject fake stack frames and pointers to them in the *data* area, using overflow and preventing segmentation faults.

4.1.1 Address Space Layout Randomization ASLR

(...)

4.2 Structural Vulnerabilties

Many attacks exploiting structural vulnerabilities were aimed to discover alive nodes in a network. There is *no control* on the fields of IP packets, thus senders are not authenticated. A threat agent may send *ECHO* messages using a broadcast address or to specific nodes to find out which are alive.

The *Distributed* ***Denial of Service*** is performed by flooding a network with *echo* messages, making the bandwidth occupied by such messages and relative replies, stopping the services running in such network.

A *Slow Denial of Service*, instead, exploits collisions in a hash table to fill up memory and slow down performance of a Module.

Is this a local vulnerability? Maybe not because it also depends by the possibility for the attacker to push input into the hash table.

4.3 Security Partial Views

4.3.1 Encryption

According to Baiardi, **Encryption** *simplifies* some **security** problems, but does **not** *solve* them. Encryption guarantees *Confidentiality*, and sometimes *Integrity*, but **not** *Availability*. Some claim that encryption solves security, but it must be taken into account that the operating system of a module may not provide a way to protect the encryption keys properly.

4.3.2 Authentication

Most security problem require three problems to be solved:

1. User identification
2. Resource identification
3. Analysis of access rights

User identification is not sufficient, *as some may say*, because it indicates which row of the **Access Control Matrix** to consider, but not how to actually use the matrix.

To provide authentication three classes are considered, and the well known two-factor authentication should pick factor from two of these:

1. *Something you* ***know***: password, PIN, pet name
2. *Something you* ***own***: smartphone, credit card, token
3. *Something you* ***have***: biometric features, i.e. fingerprint, voice, retina

Chapter 5

Discovering Vulnerabilities

5.1 Classification

To address the problem of finding out vulnerabilities many classifications have been proposed, and each one has its own purpose: module affected, how to discover it, enabled attacks, ...
It is mandatory to understand a classification goal before using it.

It is possible to consider where the vulnerability resides to provide some sort of classification:

1. **Procedural**: the actions executed are not correct
2. **Organization**: actions well defined but wrongly executed
3. **Tool**: actions well defined and correctly executed but by bad tools, e.g. OS, compiler, run time support...
   Password transmitted in clear, missing checks on boundaries...

About **tool vulnerabilities**, it is important to pay attention to **code reuse**. It must be considerend that reusing code may mean re-enable a vulnerability in such code. Code reuse is ok, but only along with **code Hardening**, which means, removing instructions and libraries which are not needed.

About the implementation, it is relevant to avoid missing controls on stuff like user input, function parameters, confused program-flow... Generally a strong type system may aid to address these kind of problems.

Besides, to avoid structural vulnerabilities, it is crucial to check whether certain modules depends on the security checks performed by others.

Searching for Vulnerabilities

Aside from the distinction between *Local* and *Emerging* vulnerabilities, it is also important to distinguish between **standard** modules (OS, web servers...) and **specialized** ones (dynamic pages produced by the server).

5.2 Vulnerability Life-Cycle

1. Born when someone does something wrong
2. Known when someone discovers the error
3. Public when its presence is revealed and it is inserted in some public database
4. Some look for a remedy/fix while others search for an exploit[1]
5. Vulnerability might become exploited
6. If existing, the fix should be applied ASAP

Note that this life-cycle doesn't take into account a **zero-day** vulnerability, which is a vulnerability not made public, whose discovery is shared only among few teams or people

Historically the most dangerous vulnerabilities are public and exploited ones; even the oldest ones are exploited because attackers are lazy and defenders even more.

[1] A program to implement an attack that exploits it

5.3 Attacker vs Owner POV

Considering the point of view of an **owner** who wants to search for vulnerabilities to improve their system's security.

In order to search for vulnerabilities, an inventory of **all** the system modules is required. This is not a trivial task, but it is necessary.

You cannot protect what you don't know.

The opposite view is the **attacker**'s one. Usually vulnerabilities of standard modules are **known**, thus an attacker may only need to know which *modules* compose the system. An attacker may acquire knowledge on the vulnerabilities from public or private (by paying) databeses, or by buying such information in the deep web. It's rare for an attacker to look by himself for vulnerabilities in a module.

5.4 Scanning

5.4.1 Fingerprinting

Active fingerprinting is a (set of) tool which exploits the fact that modules communicate through ports, and appears quite appealing from both the mentioned views: given a range of IP addresses, it sends packets on each port and analyzes the replies to *fingerprint*[2] the module listening on the port.
The *owner* might want to run the tool on the entire network, while an attacker may target single (or a few) hosts at a time.

Active fingerprinting is noisy and might considerably slow down the network performance, which in some systems, e.g. *ICS* (*Industrial Control System*), must be avoided.
(TODO - CAPIRE MEGLIO)

A **Passive fingerprinting** does not imply direct interaction with modules, but acts as a **sniffer**, analyzing packets the modules exchange in a transparent way. It exploits info in TCP and IP headers to fingerprint modules. The counterpart is that in networks with low noise/packets exchange, passive fingerprinting may take long to discover all the features of interest. Usually cannot be used by attackers.

It is important to note that a scanner may not know whether a **patch** has been applied or not to a module, hence it may report vulnerabilties which in fact have been patched, generating a **false positive**. A scanner may also generate **false negatives**, since some vulnerabilities for a given module may not appear in the DB the scanner uses to map modules to related vulnerabilities. There are also some **breach and simulation** tools which besides scanning also execute an exploit to check whether given vulnerabilities have been patched or not. Even though interesting, it may be dangerous to run such software in low-tolerance systems.

To evaluate vulnerabilities discovery methods it is common practice to use a **confusion matrix**[3], which provides, amongst others measures, *accuracy*, *precision*, *recall*[4], *specificity*.

5.4.2 Stealth scanning

Clearly, the owner is interested in discovering if anyone aside from him is currently scanning the system. An attacker may configure message frequency and the number of nodes to scan, to reduce the chance to be detected.

5.4.3 More on scanning

An owner may combine:

- External vulnerabilities scan: Try to access the sytem from outside, to understand what can an attacker discover before starting an intrusion.
- Internal vulnerabilities scan: Aims to test and fingerprint devices and modules inside a network. It might be ran by either owner or attacker after the initial access.
- Intrusive scans: i.e. breach and simulation. These are the most stringent scans, but may be disruptive.

Anyway note that it is crucial to **periodically scan** a system, due to its eventually mutable nature but way more importantly, because about 20 new vulnerabilities get published every day, along with new potential attackers and

[2] i.e. "discover the identity of"
[3] wikipedia.org/wiki/Confusion_matrix
[4] sensitivity

attack techniques.

5.5 Searching in a Module

Vulnerabilities can be searched and assessed when designing the system. Modules can be standard (e.g. OS), and thus be affected by public vulnerabilities, or specialized modules, whose vulnerabilities are unknown.

2 - Ottobre

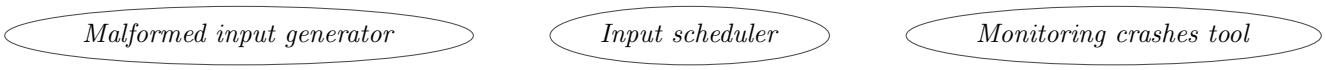
Static Application Security Testing (SAST) indicates tools of static analysis, which has advantages like scalability and easy patching, but is limited on other aspects, such as:

- ◇ Authentication Issues
- ◇ Dangerous Management of access rights
- ◇ Unsafe use of cryptography
- ◇ Many False negatives
- ◇ Cannot evaluate runtime values

Opposed to static analysis, there are *Dynamic Analysis* techniques on which big companies generally rely on. The most common technique is **Fuzzing**.

5.5.1 Fuzzing

Fuzzing is associable to *"Chaos Monkey"* test design paradigm: the idea is to send *malformed inputs* to a module. If the system responds with a crash to malformed inputs, such crash may indicate a bug, i.e. a *vulnerability.*
A fuzzing tool results from the composition of three modules:



Before executing a fuzz test, a **tainting analysis** is perfomed. Its aim is to computer for each input the set of variables that could be affected by the input. Usually this is done along with a coloring analysis, i.e. *TODO.*

(...)

Black-box fuzzing makes no assumptions on the implementation details of modules. It has grown a lot in popularity since it is faster and, more importantly, nowdays many modules, especially in IoT sector, do not provide open source code.

Some rules to efficiently apply fuzzing should be considered: First of all, input-format knowledge should be kept in mind especially for black-box debugging; in general the longer you run a test, the more bugs you may find, until a saturation point is reached; It is advisable to use different fuzzers, since they may find different bugs.

5.6 Web Vulnerability Scanner

Despite the name, it doesn't work as other scanners: its focus is onto discovering vulnerabilities in a website whose behaviour is determined by the dynamic generation of pages. Passwords and credentials may be given as input to look deeper into the website. Such scanners work much more like to *breach and simulation* then to regular scanners.

- ◇ SQL Injection: inserting or deleting information from a database
- ◇ XSS: inserting a malware on a website to be later downloaded and executed by the end user (*cross site scripting*)
- ◇ CSRF: forces an end user to perform unwanted actions on a website he is authenticated. (*cross site request forgering*)

3 - Ottobre

5 - Ottobre

Chapter 6

Attacks

3 - Ottobre

6.1 Attacks and Vulnerabilities

Following the discovery of a vulnerability v there's an analysis to evaluate which attacks are enabled by v. **Attacks** can be described as a set of attributes:

1. Precondition
2. Postcondition
3. Success Probability
4. Know how, abilities, tools required
5. Noise = Probability of being discovered
6. Automated/Potentially automatable/manual
7. Local/Remote
8. Actions to implement attack[1]

Even though some attack evaluation proposals map to each attribute a number and combine them into a value, such evaluations do not consider that **risk** resides in *intrusions*, not individual attacks, because they have a considerable impact on the system, and keep in mind that are composed by:

- ◇ Exploration and information collection
- ◇ Persistence
- ◇ Attack *chain* for privilege escalation

6.2 Attack Classification

The actions need to implement an attack may be used to define a **taxonomy** of attacks:

1. buffer/stack/heap overflow
2. *sniffing* → Illegal access to info in travel
3. *replay attack* → Repeated exchange of legal messages
4. *Interface attack* → Illegal order in the invocation of API functions
5. *Man-in-the-middle* → Interception and manipulation of info in travel
6. Diversion of an information flow
7. *Race-condition* → Time-to-use time-to-check
8. *Cross site scripting* → XSS
9. SQL injection
10. *Bell-Lapadula policy* → Covert channel
11. Masquerading as
    - ◇ user
    - ◇ machine (*IP/DNS spoofing, Cache poisoning*
    - ◇ connection (*connection stealing/insertion*)

[1] See following Section on attack taxonomy

6.3 Examining attacks

Replay attack

Suppose a user asks the bank to transfer some money to Y account with an M message. Y may sniff and record M, and before the secure channel S gets deleted, Y sends M several time.
Note that the attack may work even if encryption is used.

Man-in-the-middle

If A and B communicate, E may pose itself in the middle, acting as if it were B to A and A to B. Such attack is possible when no authentication is required.

XSS

A website allows users to upload contents to be later (possibly) downloaded by users. Thus a malicious user may upload hidden scripts to damage or steal information from the user who download their content. To avoid this the website must check the content uploaded by users.
A well known attack of this type targeted BBC.

SQL Injection

An input may insert a malicious query (i.e. `DROP TABLE USERS`) in a credentials field. The best way to avoid this is to whitelist using RegEx.

Cryptography attacks

These are a category on their own, there are many types, with different variations and features.

Side-channel attacks

Any attack that measures some physical value to discover an encryption key. Currently it is popular due to the capabilities of machine learning in exploiting large number of pairs to deduce a function.
Such measures may be:

- Electromagnetic emissions
- Energy consumption
- Execution time to discover inner status
- Execetion time to discover cache usage and prediction mechanisms.

Virtual Machines & Blue Pill

Cyber system may be composed of many virtual machines onion-like organized. Thus, attacking a low-level VM may grant access rights to higher ones.
Besides, an attacker may insert a new VM in the hierarchy: this is called *Blue Pill* attack, it's hard to discover and has a high impact. A new VM may return to higher VMs fake information on the status of the underlying machines and/or send malicious commands to the underlying machines.
Stuxnet was a malware which used to send commands to uranium enrichment centrifuges to destroy them, and meanwhile told the operator that everything was going well.

Chapter 7

Patching

3 - Ottobre

5 - Ottobre

7.1 Patching

Patching is **slow** and **expensive**. This is due to many factors: first of all there's the need to run **regression** tests, to check correctness of standard behaviour and of he bug to be corrected; besides, new behaviours and problems may arise because of the new code; in case of a complex problem requiring N patches, the **scheduling** of such patches must be taken into account; in general is advised to patch using an automated process exploiting patching agents and environment.

7.1.1 Common Vulnerability Scoring System

Note that, for instance, *Industrial Control Systems* **cannot be patched**, because it would imply for the production to be *suspended* and for the whole system to be *certified* again.
This forces an admin to decide whether *"to-patch or not to-patch.*
About this matter a **Common Vulnerability Scoring System** (*CVSS*) has been developed. Its aim is to consider the main features of a vulnerability and compute a score based upon them; in the initial idea such score should have allowed to define a score threshold to decide whether to patch or not. However, such idea doesn't work for two main reasons:

1. Single vulnerabilities are not of interest, while *intrusions* are i.e. chaining and exploiting multiple vulnerabilities
2. *CVSS* totally *ignores* the **system**, but the context in this topic is fundamental

Even if it cannot be helpful as a guide for an admin to perform the above mentioned decision, it can be truly instructive to understand how impactful a vulnerability can be, and it can provide an approximation of how *difficult* may be for an attacker to exploit such vulnerability. The CVSS provides three *metrics* to evaluate vulnerabilities risks:

- ◇ **Base** fundamental characteristics constant over time and user environments. Such metric aims to provide an intuitive and clear vulnerability representation.
- ◇ **Temporal** the characteristics that change over time but not among user environments
- ◇ **Metric** the characteristics relevant and unique to a particular user environment

...Specs on metrics...

7.1.2 CVSS revisions

Dragos in 2022 proposed a revision of scores in the CVSS considering the attacker point of view, claiming to have better ones, but still raising up many doubts.

Later on the same experts team created the **EPSS** as a measure of exploitability: it is a *Neural-Network* based system which estimates the probability that a vulnerability will be exploited. It is unclear on which data the AI has been trained, accuracy, tuning, etc... EPSS does not consider risk, context or whatsoever, it's just a probability estimator.

SSVC *Stakeholder-Specific Vulnerability Categorization* aims specifically to produce an **action**. Imagine a decision tree 5 levels deep. An admin must make 5 "decisions"[1] about a vulnerability, and then a leaf of the tree can be one among 4:

1. Track
2. Track
3. Attend
4. Act

Considering the decisions to be made by the admin:

1. **State of Exploitation**
   - ◇ *None*: No evidence of atcive exploitation and no public *Proof of Concept* (PoC) on how to exploit the vulnerability
   - ◇ **Public PoC** Sites like ExploitDB or Metasploit contain PoC on such vulnerability v or v has a well-known method of exploitation
   - ◇ **Active** Credible sources claim that v is shared, observable and has been exploited in the past.
2. **Technical Impact**
   - ◇ **Partial** control of the software given to the attacker e.g. DoS attack
   - ◇ **Total** control of the software or total information disclosure given to the attacker.
3. **Automatable**
4. **Mission Prevalence & Public Well Being**
   - ◇ Does the vulnerable component provide support for the (attacker?) mission? Is it essential? Is it not so useful?
   - ◇ Which kind and how much harm the attack may cause and if it is irreversibile or not. Physical, psychological, financial, environmental
5. **Mitigation**
   This is not included in the decision tree!
   - ◇ Fix available/unavailable
   - ◇ System change difficulty
   - ◇ Actual Fix or Workaround

[1] In the sense of answering 5 questions

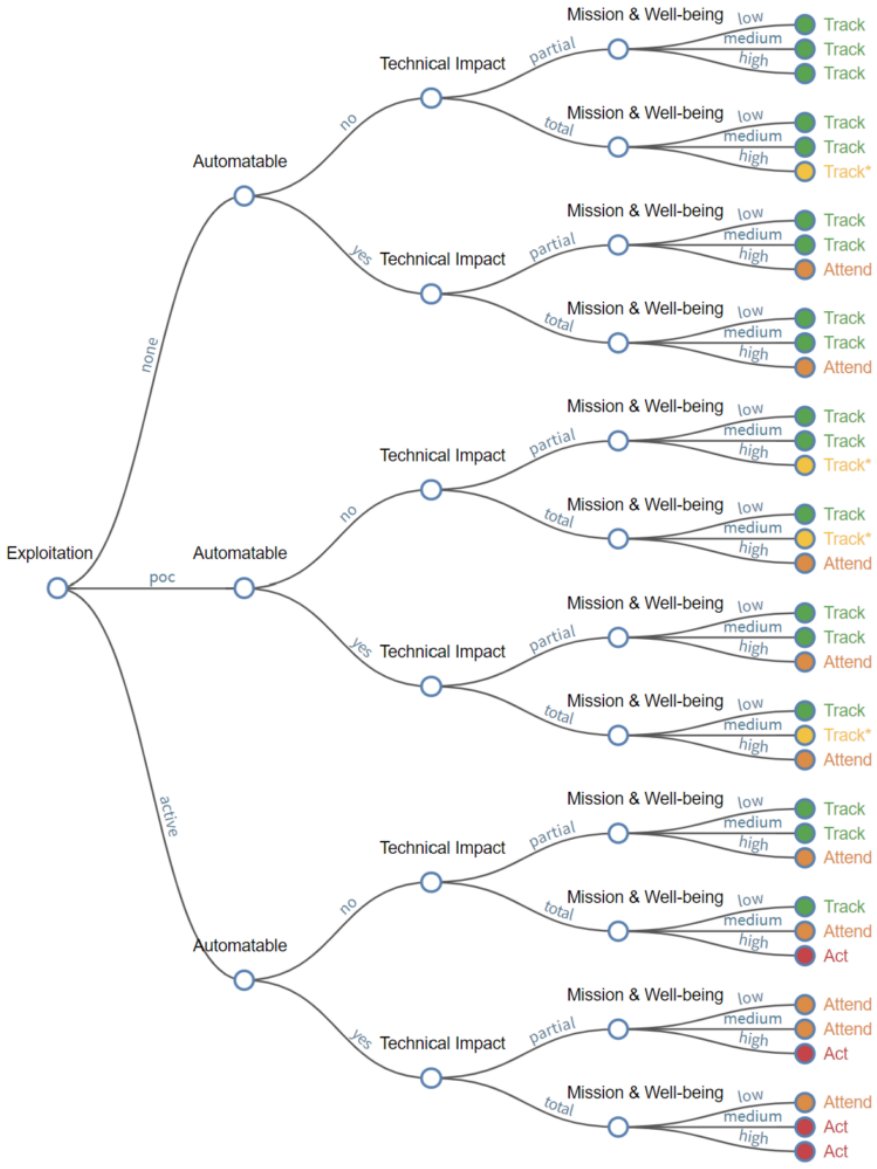


Figure 7.1: SSVC Decision tree

Chapter 8

Countermeasures

10 - Ottobre

8.1 Introduction

- ◇ **Proactive** Patching a vulnerability before being vittim of an attack
- ◇ **Dynamic** Countermeasure applied during an intrusion to prevent the attacker from reaching its goal e.g. dropping connection
- ◇ **Reactive** Patching applied after an intrusion to prevent the success of the next one.

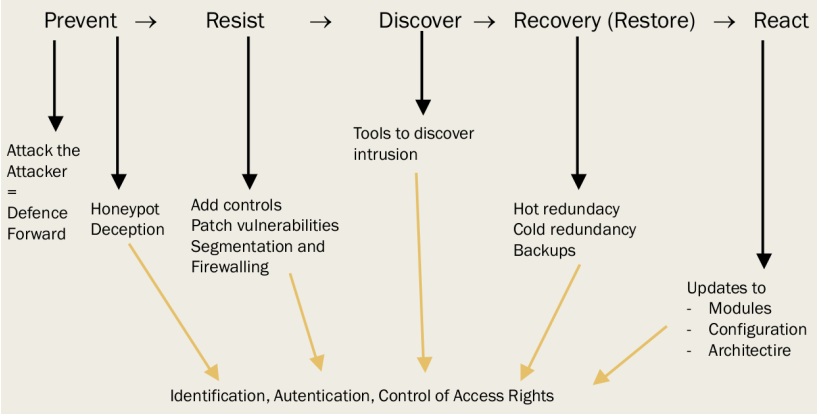


Figure 8.1: Countermeasure more detailed classification

8.2 Robustness and Resilience

Robustness refers to strength and effectiveness, even in adverse conditions. The more robust a product is, the less its performance is affected by disruptions or input changes, because such changes have been predicted and contingency plans have been developed and built into the product.

Resilience instead is the ability to bounce back after disruption. Unlike robustness, which is proactive, resilience is reactive, following incidents in which system performance has already been affected. Resilience is measured in terms of the time a system takes to recover to its original state of performance or better. Both can exploit **redundancy** and **heterogeneity**; **heterogeneity** means using modules from distinct suppliers to avoid a catastrophic failure due to a single vulnerability, increasing robustness.
Redundancy can clearly highly increase *resilience* to faults, there are many kinds of redundancy:

- ◇ *Cold* redundancy: spare and idle instances of modules are started only when working instances are unavailable due to faults or attacks.
- ◇ *Hot* redundancy: multiple active instances that work simultaneously to tolerate loss due to attack without a recovery e.g. multiple DB copies, or nodes in a network/cloud
- ◇ *Triple Modular* redundancy: hot redundancy where three instances of a module receive the same input, execute the same computation and vote the result. Vote can be decentralized or centralized. Space systems use five

copies.
Increases safety but maybe not security
- ◇ Overall oversized system in general may allow resource loss

Anytime resilience is based upon reconfiguration, system monitoring is fundamental. System monitoring should discover how close the current system behaviour is close to a boundary and fire the reconfiguration action to remain or return to normal behavior. The largest the amount of information on the current behavior, the higher the performance of monitoring.

8.2.1 Minimal system

One way of dealing with intrusion is to has a **minimal system**, i.e. a subset of the system of robut and heterogenous modules, which can act as a starting point to restore a consistent status. It is crucial not to lose control on the minimal system, since doing so might mean not being to restore the normal behaviour.

From this point of view we can also consider a **minimal behaviour**, i.e. the smallest set of behaviours that is acceptable for the final user. The minimal behaviour requires some features to ensure robustness which can also be used to restore the normal behaviour after an attack.
Consider an ATM and its behaviours as an example, and notice which behaviours compose the minimal one:

1. Protect money
2. Interact with a central system
3. Distribute money
4. Distribute information about accounts

8.3 Authentication

$$\langle subject, object, operation \rangle$$

Keeping in mind this triple, there are two kinds of controls which can be done on it; subject *identity* controls and *access rights* ownership controls (i.e. the subject owns the access right).

Most of the mapping of identity into a set of access rights is a task that usually is delegated to the OS, which can also handle authentication, but typically specialized components are preferred.

Authentication can be classified as follows:

- ◇ Weak Static: passwords and similar strategies which can be easily defeated by a sniffing atacker
- ◇ Weak not Static: cryptographic techniques to produce information that is not repeated
- ◇ Strong: mathematics and encryption to produce information that is not repeated and that may be validated by a distinct channel

8.3.1 Authentication Mechanisms

1. Something the user knows e.g. a password hashed on the server
2. Something the user owns
3. Something the user "is" e.g. Biometric Authentication through fingerprint/retina/face

8.4 Kerberos

Strong authentication network protocol

- ◇ A user password must never travel over the network
- ◇ must never be stored in any form on the client machine
- ◇ must be immediately discarded after being used
- ◇ should never be stored in an unencrypted form even in the authentication database

A user enters a password only once per session so that it can transparently access all the services it is authorized for without having to re-enter the password during this session. Authentication information management is centralized on the authentication server. Application servers must not contain authentication information. Such centralization guarantees no redundancy and possible consistency problems, allows an admin to perform edits on the auth DB in a one-time action.

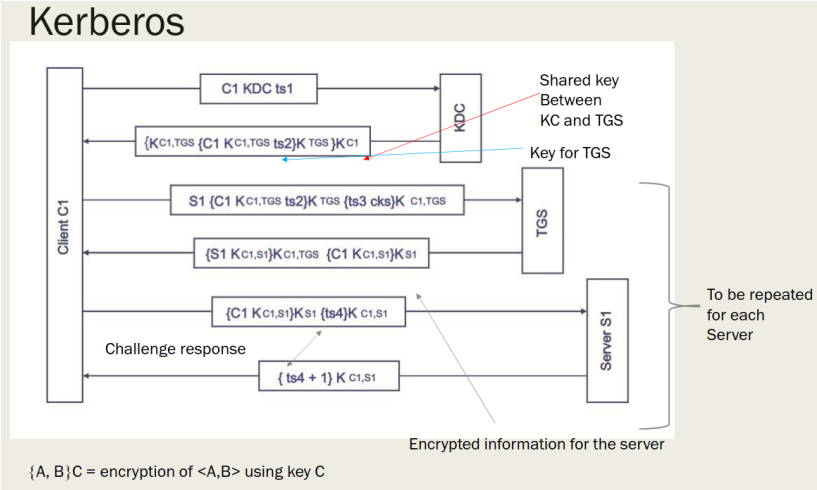


Figure 8.2: Kerberos messages exchange

Kerberos provides a three-sided authentication with a shared key for **symmetric encryption**. The agents are the following:

1. Client
2. Server
   offers a service but want the users to be authenticated
3. KDC Key Distribution Center
4. TGS TIcket Granting Service

Base Principle: *If you know the shared key then you have been authenticated by the KDS*
A **ticket** allows a client to prove its identity to a server to access the service one offers. A ticket is valid in a time window only. However, the **counterpart** is that centralization introduces both *single-points-of-failure* and potential *performance bottleneck*.
Master Keys are encrypted with the *master key* of the **KDC** and with the one of **TGS**. The passwords of these two modules are the last line of defence.

8.5 Zero Trust

A key point of zero trust is that authentication and authorization involve both a subject and a used *device*. The focus of zero trust is to protect resources instead of network segments since the network location is no longer seen as the prime component of the security posture of the resource.

8.6 Control and Management of Access Rights

As said before access rights are represented as a matrix which is highly dynamic.
Note that any OS has an implementation of this matrix to protect physical resources, logical resources and memory areas. Besides each application may have its own matrix to protect the resources it manages.

Basically, an OS matrix determines which users can interact with a given application, and also which operations such users can invoke.

A whole matrix is inefficient due to centralization, thus it is advised to decompose on *rows* or *columns*.

8.7 Rows - Capabilities

A possible implementation is the one with capabilities: this solution stores the access rights in the subject that then extracts and presents the one that enables the operation of interest. An example of usage is the map that translates virtual addresses on physical addresses, each entry in this map contains some bits that represent the permission for that memory region.

On **distributed** systems, we need some way to protect capabilities information from tampering because hardware features can't certify that. To solve the problem we use a shared secret among the systems and we use that secret as

a key to build check digits that are a hash of the capability produced using the key. The receiver of the capabilities and of the check digits can use the digits to verify the capability is authentic.
In **centralized** systems, instead, each pointer corresponds to a capability.

8.8 Cols - Access Control List

This solution provides storing the matrix in colums, one column for each object. An object implementation also stores the information to control the accesses to the object.

An instance of ACL is the packet routing in Linux made with `iptables`, which allows to define rules for each packet "chain":

- ◇ Input chain
- ◇ Output chain
- ◇ Forward chain

Besides it is possible to define default policy PASS/DROP. For each packet it is possible to perform various actions:

- ◇ DROP/PASS (route)
- ◇ goto/return i.e. call/return packet to a chain
- ◇ Queue i.e. handle packets queue using user's code
- ◇ Log
- ◇ Reject
- ◇ dnat/snat/masquerade

In general, filtering packets going *outside* a node, is called **egress filtering**; in other words, before allowing an *outbound connection* a user-defined *rule* must be checked. Such control allows to *discover malware*, stop contributing to *attacks*, or to *block local users* from using illegal services.

8.9 Role Based Access Control

Basically pairing *access rights* with a *professional* ***role***. When representing such access rights, there is a simple matrix for each role, making room for scalability and easy management. Clearly, also a mapping between users and roles is needed.

Roles may be partially ordered, leading to $R_1, R_2 \Leftrightarrow (R_2.access\_rights \in R_1.access\_rights)$

8.10 Attribute Based Access Control

Access rights are granted or denied according to values of 4 key attributes:

1. ***Subject***
2. ***Action***
3. ***Object***
4. ***Contextual***

This is a flexible approach, but strongly relies on how attributes resist manipulation. Besides it must be noted that *ACL* and its derivations can grant access even to unknown subjects, which instead is not possible with capabilities since they are distributed only to entities we already know.

Chapter 9

Windows authentication

The key concept is the relationship between **logon sessions** and **access tokens**. A logon session represents the *presence* of a user on a machine and begins with a successful authentication and ends when the user logs off.

When a user logs in they provide a pair of $\langle username, password \rangle$ which is checked by *Local Security Authority* (LSA). If the credentials are valid, LSA will create a new logon session and produce an **access token**; multiple access tokens may be associated with a session, but one token can only be linked to one session, typically the logon that generated it. However Windows can change the logon session (and cached credentials) a current token is associated with.

9.1 Access token

Access tokens cache some attributes regarding the user security context, i.e. the privileges and permissions of a user on a specific workstation (and across the network).

- ◇ The security identifier (SID) for the user
- ◇ Group memberships
- ◇ Privileges held
- ◇ A logon ID which references the origin logon session

An *access token* act as proxy or stand-in for the logon session. When making security decisions, Windows never interact with the logon session itself ("hidden" in `lsass`, the process implementing LSA), but with an access token which represents it.

9.1.1 Mandatory Integrity Control

Aside from access tokens, there is another secury level in Windows: security *principals* and securable *objects* are assigned **integrity** levels that determine their levels of protection or access. MIC is a mechanism for controlling access to securable objects in addition to discretionary access control and evaluates "integrity" access before evaluating access checks via an object's DACL[1]. For instance, a principal with a low integrity level cannot write to an object with a medium integrity level, even if the DACL of the object allows write access to the principal.

9.1.2 Access Control List(s)

ACLs are the lists in a **security descriptor** with information on actions users, groups, or objects can perform on the file or folder to which the descriptor is applied.
A security descriptor may contain different two types of lists:

1. **DACLs** Discrestionary ACL - the list of SIDS for the users, groups, and computer objects allowed or denied access to perform actions on files or folders
2. **SACLs** System ACL - the list of SIDS for the users, groups, and computer objects for which successful or failed auditing events are logged

ACEs are individual entries in either DACLs or SACLs for particular users, groups, or computer objects

[1] D? Access Control List

9.1.3 Sandboxing Tokens

Applications e.g. browsers, have historically been victims of attacks. An attacker who successfully exploits a browser, then the attacker's *payload* shares the same *access token* of the browser, allowing it to perform any action the browser is allowed to do.

To mitigate such kinds of attacks, browsers' code has been moved into lower-privilege processes by creating a smaller and restricted security context; in the Unix Documentation, such context is called a **sandbox**.

The key idea is to limit the extent of an attack to only the resources accessible to the sandbox maliciously exploited.

9.2 Remote Hosts AC

A logon session is unique to a workstation and users cannot send an access token over the wire because it would be meaningless as it does not correspond to a valid logon session on the remote host. Furthermore, this is a target for replay attacks. Thus, the user needs to **re-authenticate** and establish a new session on the remote host.

In order to establish a new logon session, the **SMB** server has to authenticate the client over the network. In Windows domains, network authentication is performed via **Kerberos** or the **NTLM** challenge-response protocol. Regardless of the auth method, network logins do not cache credentials and this token cannot be used to authenticate to another remote host. This is the *"double hop"* problem.

Kerberos is **default** authentication method today, NTLM acts a backup in case Kerberos authentication fails.
In NTLM, passwords stored on the server and domain controller are not ***"salted"***, which means adding a random string of characters is not added to the hashed password to further protect it from cracking techniques. Besides NTLM doesn't support many modern encryption algorithms and techniques.

9.2.1 MS Kerberos and MIT Kerberos

In a standard authentication, a user asks its Kerberos key distribution center (KDC) for a session ticket for a specific host. In Windows instead, once authorized to enter, the user must still show his rights for the resource requested, such as a shared file or network printer. In this way the user's security access token in the application-specific data field in a message protocol

9.3 Impersonation/Delegation

In multi-threaded applications, complex race conditions may arise if different threads start enabling/disabling different privileges or modifying default token DACLs. By default all threads will inherit the same security context as their process's primary token. However, impersonation allows a thread to switch to a different security context. Impersonation enables threads to have their own local copy of a token: an **impersonational token**. Such process allows, for instance, an SMB server to handle each incoming request in a separate thread and *impersonate* the access token representing the *remote client*. Thus, **locally**, since the thread is associated with an impersonational access token, any *access checks* will be performed with such token.

What does this mean?
"as this impersonated token may be linked to a different logon session with different cached credentials the thread's security context remotely is also different"

Unless some mechanism protects the token, a thread running as SYSTEM can modify it. To avoid too impactful exploitation of such feature, there is an undocumented feature called *"trust labels"*, which is an optional component of every security descriptor, restricting specific access rights to some types of protected processes.

Chapter 10

Deception

10.1 Honeypot

A **honeypot** is a system designed exclusively to be attacked an to **collect information** about the attacker and its tactics, techniques and procedures; the other focus of an honeypot is also to possibly slow down an attack to a system by **diverting** it on itself.

The scaled version of a honeypot, is a **honeynet**, which is an entire network attached to a real system designed to be targeted instead of the main system.

10.1.1 Classification

1. **Interaction**-based
   i. *Low* - e.g. simple port listener
   ii. *Medium* - emulation of a network service that analyzes the inputs and returns some replies similar to those the real service would return.
      ◇ Simulates just some **features** of the service
      ◇ Easy to implement, **low risk**
      ◇ Can collect a low amount of information
      ◇ Tools ⟶ OS + Honeyd
   iii. *High* - built around real services that run on real machines to fool the attacker
   realistic but dangerous due to the large amount of vulnerable software
      ◇ Simulates **all features** of the service and of the underlyng OS
      ◇ The attacker may fully compromise and control it
      ◇ **High risk**
      ◇ A larger amount of information
      ◇ Tools ⟶ Honeynet.
2. **Implementation**-based
   i. *Virtual*
   ii. *Physical*
3. **Goal**-based
   i. Production
   ii. Research on attacker behaviour

10.2 Honeyd

Honeyd is a daemon which creates virtual nodes in a network. It is highly configurable and is able to reproduce even large and complex networks; besides it can integrate with virtual and physical real-existing networks.
Honeyd provides many features, we can list some of the main ones:

- ◇ It detects illegal activities in a network by monitoring the IP addresses that are not withing a **range** named *"dark space"*. Any attempt of connection to or from the *dark space* is assumed to be an attack or a vulnerability scan.
- ◇ It monitors activities related to TCP and UDP ports and ICMP traffic.
- ◇ It can emulate network services using script in Perl, shell or other way of interacting with the attacker.

10.2.1 Architecture

- ◇ *Configuration database* - Queried to discover the model paired with the destination IP address
- ◇ *Packet dispatcher* - analyzes input packets and checks correctness and integrity. Anything different from TCP, UDP and ICMP gets *discarded*.
- ◇ *Protocol manager* -
- ◇ *Personality engine* - computes a reply packet and updates it to guarantee coherence with the OS that the destination is expected to use
- ◇ *Optional routing component* - allows the routing of a packet to a real application

10.2.2 Research results

Honeypots provided an important amount of data to perform research on. Many statistics have been computed to produce estimations and interesting results, a ***UniPi*** *student presented a thesis on the topic* ☺.

Chapter 11

Countermeasures

20 - Ottobre

11.1 Robust Programming

Ideally it indicates a programming style focused on minimizing vulnerabilities and the impact of any vulnerability still exploitable.

Robust programming can be summarized with a few guidelines:

1. *Validate* program inputs aka input is evil
2. *Prevent* buffer overflow aka check sizes
3. A robust implementation minimizes any *information leaked outside* e.g. module, object, function ...
   - ◇ Logical pointers rather than physical ones
   - ◇ Validate any information that is exchanged
4. Check values transmitted to other functions (egress filtering)
5. Check returned results

Besides, it is important to focus also on **interaction controls**, robustness must be enforced on both malicious and erroneous behaviour.

11.1.1 Input validation

Usually input validation is achieved with a form of *default* ***deny*** by defining a legal input structure and discarding every input which doesn't satisfy it.

In case of string this may be done through RegEx, max length, ...

It is important that the checks to validate the input should be specified when the program is designed rather than after an attack; besides a check should be designed in an simple and readable way, to easily ensure its correctness. Some examples of input which usually must be validated are:

- ◇ Environment variables
- ◇ File names (blanks, .., /)
- ◇ Email addresses
- ◇ URL
- ◇ HTML headers/body
- ◇ Data

Memory allocation and strings length is a crucial aspect: only library functions with an explicit string length specified should be used[1], and in general, it is appropriate to allocate only the memory actually needed by a data structure according to its sizeto avoid leaving space to store dangerous values or inputs.

Speaking of functions, attention must be paid to a rigorous **interfaces definition** and to avoid making assumptions on relationships between input and output values of function; in other words, if a function A takes as input the value

[1] e.g. `strncpy()` instead of `strcpy()`

x returned by B, it must not be *asserted* that x is for sure a **valid** value, B should check the correctness of the input regardless of knowing how it was generated.

11.1.2 CWE - Vulnerabilities Ranking

This article by ***CWE*** (*Common Weaknesses Enumeration*) lists the most dangerous and frequent software weaknesses of 2023, based on data provided by *NIST*.

The scoring formula to calculate a ranked order of weaknesses considers the **frequency** a CWE is the root cause of a vulnerability with the **severity** of its exploitation. Both frequency and severity are *normalized* relative to the minimum and maximum values seen. **Frequency** is obtained by counting weaknesses occurences in the *National Vulnerabilities Database* (NVD), while **severity** is the average computed on the Vulnerabilities score in the *CVSS*[2] a given weakness is mapped to.
The **final weakness score** is computed by multiplying frequency and severity scores.

Biases and limitations

There are two biases which CWE doesn't take into account, which somehow negatively affect how valid CWE's scores are:

1. **Metric bias**
   i. Indirect prioritization of implementation faults over design flaws
   ii. Prefers frequency over severity due to distributions of real-world
2. **Data bias**
3. i. Only uses NVD data based on publicly-reported CVE Records
   ii. Many CVEs do not have sufficient details to assign a CWE mapping, omitting them from ranking
   iii. There may be an over-representation of certain programming languages, frameworks, or weakness-detection techniques

There also a few aspects which this scoring system cannot represent and should be taken care of. First of all, weaknesses that are rarely discovered will not receive a high score, regardless of the consequence of an exploitation. Weaknesses that begin with a root cause of a mistake leading to other mistakes, create a chain relationship. As we have seen, chains of mistakes/vulnerabilities/attacks are a key point in security, but CWE's scoring system treats any $\langle V_1, V_2\rangle \wedge V_1 \rightarrow V_2$ as if V_1 and V_2 were independent i.e. $V_1 \not\rightarrow V_2$.

11.2 Firewall

A **firewall** is a module to filter all the messages exchanged by *two* networks with a distinct security level; *all and only* the messages travelling on the wires connecting the two networks cross the firewall and therefore get filtered. A firewall works correctly under the assumption that a network has been split (*segmented*) into two **subnets**, and that it *correctly implements* a security policy, which should ***not*** define the policy by itself. Firewalls are usually **classified** on the known and manageable **protocols** and on their **implementation**.

11.2.1 Segmenting

Firewalling goes along with **segmenting** a network, which results in multiple subnets with different security levels whose interaction is determined by firewalls inbetween them. This architecture increases **robustness** by preventing an attacker from having **initial access** on an entire system and from freely performing **lateral movements**; besides this architecture perfectly integrates with **honeypot** deception mechanisms.

11.2.2 Classification

Firewall may operate on different levels of the TCP/IP stack and in different ways:

- ◇ Packet filtering firewall
- ◇ Circuit-level gateway
- ◇ Application-level gateway (aka *Proxy Firewall*): firewall which recognizes application level protocols and can make assumptions on it
- ◇ Stateful inspection firewall *Stateful* means that the firewall inspects also the contents of a communication and the properties related to the status of a connection.
- ◇ Next-generation firewall (NGFW)

[2] *Common Vulnerabilities Scoring System*

At level 3 (IP Packet Inspection) the firewall can check only the header of IP packets, while at level 4 (circuit level firewall) ...

TODO

11.2.3 Pros & Cons analysis

Packet-filtering firewall

Pros
- A Single device can filter traffic for an entire network
- Extremely fast and efficient in scanning traffic
- Inexpensive
- Minimal effect on other resources, network performance and end-user experience

Cons
- Since traffic filtering is entirely based on IP addresses and ports, it lacks broader context that informs other types of firewalls
- Doesn't check the payload and can be easily spoofed
- Not an ideal option for every network
- Access control lists can be difficult o set up and manage

Circuit-level gateway

Pros
- Only processes requested transactions; all other traffic is rejected
- Easy to set up and manage
- Low cost and minimal impact on end-user experience

Cons
- No protection against data leakage from devices within the firewall that should be used in conjunction with other security technology
- No application layer monitoring
- Requires ongoing updates to keep rules current

Stateful inspection

Pros
- Monitors the entire session for the state of the connection, while also checking IP addresses and payloads for more thorough security only layer 4 information
- Offers a high degree of control over what content is let in or out of the network
- Does not need to open numerous ports to allow traffic in or out
- Delivers substantive logging capabilities
- Some defence against DOS

Cons
- Resource-intensive and interferes with the speed of network communications
- More expensive than other firewall options
- No authentication capabilities to validate traffic sources aren't spoofed

Most popular firewall as it acts as a gateway between computers and other assets within the firewall and resources beyond the enterprise

Application-level

Pros
- Examines all communications between outside sources and devices behind the firewall, checking not just address, port and TCP header information, but the content itself before it lets any traffic pass through the proxy. Layer 7 analysis
- Provides fine-grained security controls that can, for example, allow access to a website but restrict which pages on that site the user can open
- Protects user anonymity

Cons
- Can inhibit network performance
- Costlier than some other firewall options
- Requires a high degree of effort to derive the maximum benefit from the gateway
- Doesn't work with all network protocols

NGFWs are an essential safeguard for organizations in heavily regulated industries, such as healthcare or finance

NGFW

Pros
- Combines DPI with malware filtering and other controls to provide an optimal level of filtering. Considers also sender addresses
- Tracks all traffic from Layer 2 to the application layer for more accurate insights than other methods
- Can be automatically updated to provide current context

Cons
- In order to derive the biggest benefit, organizations need to integrate NGFWs with other security systems, which can be a complex process
- Costlier than other firewall types

NGFWs are an essential safeguard for organizations in heavily regulated industries, such as healthcare or finance

Proxies

Proxies protect clients from attacks from an external server, while **reverse proxies** protect internal servers from attacks by external agents, besides they can also act as a load balancer.

11.2.4 Wrapping Up

Stateful inspection is the most common technology: it works at the network layer and provides dynamic packet filtering. While packet filtering examines information in a packet header, stateful inspection tracks each connection traversing any firewall interface and confirms they are valid. It is a system backed up by a state table that tracks all sessions and inspects all packets passing through: if packets have the properties the state table predicts, they can pass, otherwise they don't. Clearly the state table changes dynamically according to traffic flow.

Feature	Packet-Filtering Firewalls	Circuit-Level Gateways	Stateful Inspection Firewalls	Application-Level Gateways (Proxy Firewall)
Destination/IP Address Check	Yes	No	Yes	Yes
TCP Handshake Check	No	Yes	Yes	Yes
Deep-Layer Inspection	No	No	No	Yes
Virtualized Connection	No	No	No	Yes
Resource Impact	Minimal	Minimal	Small	Moderate

Figure 11.1: Brief Firewall families comparison

11.2.5 Takeaway guidelines

These are the guidelines according to SNAS which indicate "suspicious" traffic **outgoing** from your network, thus the network traffic which should be **egress-filtered**.

- ◇ All traffic directed to IP addresses in your network(or that you manage)
- ◇ MS RPC (TCP/UDP 135), NetBIOS/IP (TCP/UDP 137-139), SMB/IP (TCP/445)
- ◇ Trivial File Transfer Protocol - TFTP (UDP/69)
- ◇ Syslog (UDP/514)
- ◇ Simple Network Management Protocol – SNMP (UDP 161-162)
- ◇ SMTP from all but your mail server
- ◇ Internet Relay Chat IRC (TCP 6660-6669)
- ◇ ICMP Echo/Reply
- ◇ ICMP Host Unreachable

11.3 Segmentation

A **segmented** network forces an attacker to adopt **pivoting**, which is attacking a host only to exploit it to route traffic to other nodes or subnets, usually this is achieved with the aid of a **beacon** to remotely control the host. Hence, in general, segmentation leads the attacker to implement *more* attacks.

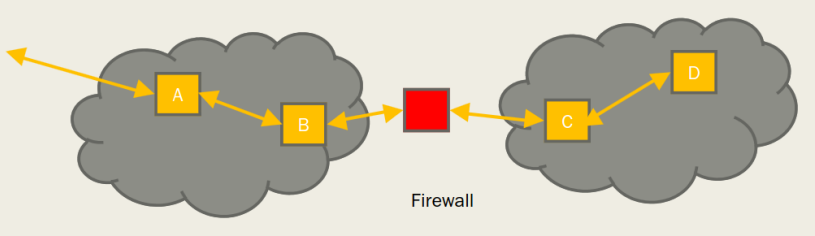


Figure 11.2: Segmented network and the need for Pivoting

The attacker needs to perform pivoting to intrude in the network, thus they need to attack at least one host in left subnet and one in the right subnet; otherwise they'd need to attack the firewall, but generally this is more costful in terms of effort and resources.

It is also possible to combine **firewalls** and **honeypots**. They can be placed either in the internal network amongst other nodes or between the router and the global/outside network.

Microsegmentation

What about **cloud**-based computing? The concept of firewalling still holds, even for virtual networks, but with a few adjustments.

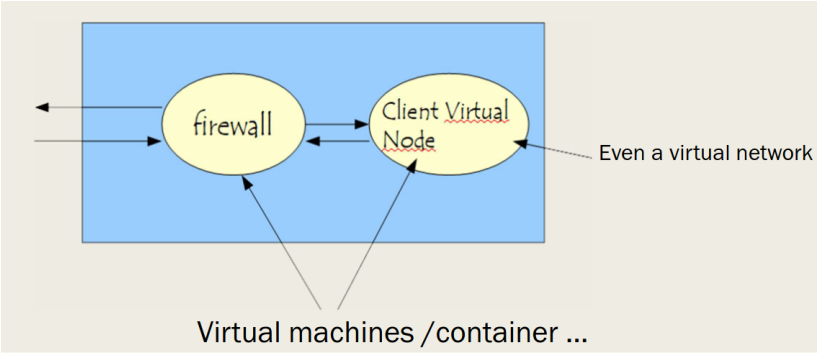


Figure 11.3: Firewalling in cloud environments

Microsegmentation software with network virtualization technology is used to create "zones" in cloud deployments. These granular secure zones isolate *workloads*, securing them individually with custom, workload-specific policies. This kind of granular security allows organizations to apply security controls to individual workloads and applications, rather than having a single security policy for the entire server.

In this scenario, we can broadly define a **workload** as the resources and processes needed to run an application. Hosts, virtual machines and containers are a few examples of workloads. Since most modern enterprise systems are distrubuted among many cloud and local architectures, the goal of microsegmentation and zero trust is to overcome **Perimeter Security** while protecting workloads.

Perimeter security makes up a significant part of most organizations' network security controls. Network security devices, such as network firewalls, inspect "*north-south*" (client → server) traffic that crosses the security perimeter and block "bad" traffic.
Assets within the perimeter are instead implicitly trusted, which means that "*east-west*" (i.e. workload to workload) traffic may be allowed without inspection; this is why **lateral movements** are usually hard to be identified.
Microsegmentation provides isolation and determines if two endpoints should access each other, hence enforcing segmentation with least-privileged access reduces the scope of lateral movements and might contains data breaches.

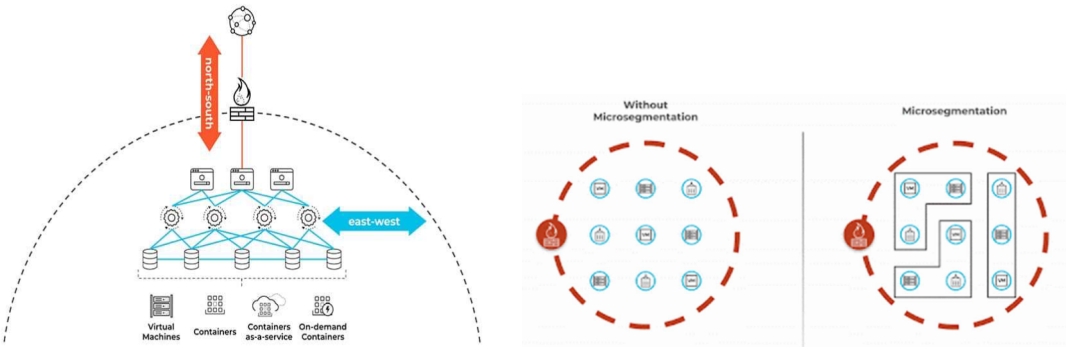


Figure 11.4: Microsegmentation example figures

VLANs and Subnets

VLANs virtually separate LANs into smaller networks, they work like normal LANs but are logically or virtually separated instead of being physically so.

Amongst reasons to use VLANs, the main one is to broadcast traffic. VLANs give us all of the benefits of physically separating our network, by doing it virtually without spending extra money on hardware and phyisical wiring management.

Subnets instead are networks inside a network or, in other words are smaller sections (subnetworks) of a larger network. Basically, subnets are a logical partition of an IP network into several smaller networks for making the network fast and efficient.

The key point is that VLAN are based on *Layer-2* protocol, while subnet on *Layer-3*.

11.4 VPN

A **Virtual Private Network** *(VPN)* is an overlay network that emulates a secure connection on top of a public (*unsecure*) network. VPN connect local subnets that may even include just one machine.

IPSEC is an IPv4 extension to encrypt an authenticate information flow. There are also other solutions to encrypt information also on different OSI layers, like PGP, HTTPs, SSL/TLS, ... There are also network boards[3] designed specifically to speed up IPSEC encryption/decryption.
IPSEC provides two possibile behaviours/protocols:

1. **Authentication** Mode: auth header
2. **Encapsulated Security Payload**: information encryption

Besides, each of them can be used in two modes:

1. **Transport** Mode: new fields are added to original packets. It is used to create a secure point-to-point connection between two nodes
2. **Tunnel** Mode: IP packets become the payload of new IPSEC packets
   This is the preferred and most popular way, since it may act transparently to internal nodes in a LAN, given that a host is delegated with the task of encrypting and decrypting

Each connection between nodes is protected using symmetric encryption while the endpoints of the VPN may own a pair of public/private keys; rhere is a protocol for the initial exchange that uses asymmetric encryption to determine the shared key to create a secure connection between two nodes that is protected through symmetric encryption. In general symmetric encryption is better than asymmetric in terms of performance, since it requires only basic operations like shifts and XOR.

IPSEC defines 4 new protocols:

1. **AH** *Authentication header* - mutual authentication and message integrity
2. **ESP** *Encapsulating Security Payload* - it guarantees confidentiality by protecting all the content that is exchanged
3. **IKE** *Internet Key Exchange* - two partners reach a consensus on the key to be used to protect their communications and on how long such key is valid 4 messages needed.
4. **ISAKMP** *Internet Security Association and Key Management Protocol* - to agree on the *"Security Association (SA)"* to be established and on its attributes.
   6 messages needed, or 3 in case of *ISAKMP-AGGRESSIVE*, which is usually preferred.

The abovementioned **security association** (*SA*) describes a direct connection with the services associated with the traffic that crosses that connection; it defines all the information needed to achieve a secure communication. The security services of a SA are implemented through either AH or ESP, even if in principle the two protocols can be applied simultaneously to the same connection this never happens in practice. Besides note that to defend a bidirectional communication two SAs are required, one for each direction.
When negotiating on the SA the two nodes exchange a *Security Parameters Index SPI* used to lookup in a *Security Associations Database SAD*.

[3] FPGAs (?)

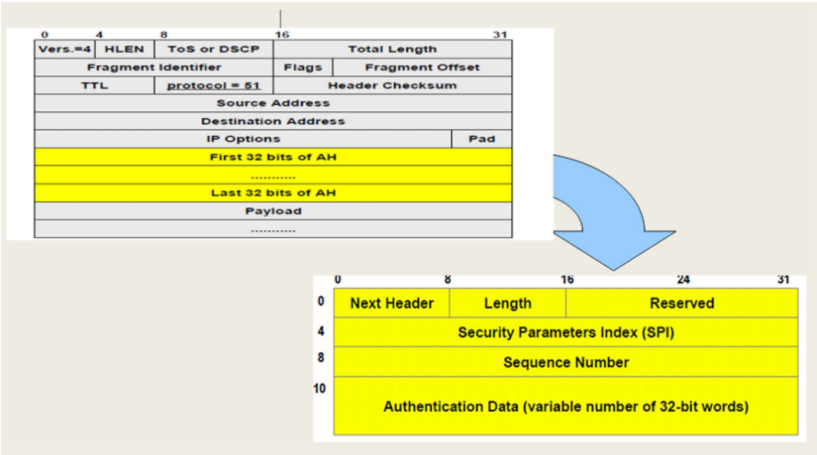


Figure 11.5: Authentication Mode header

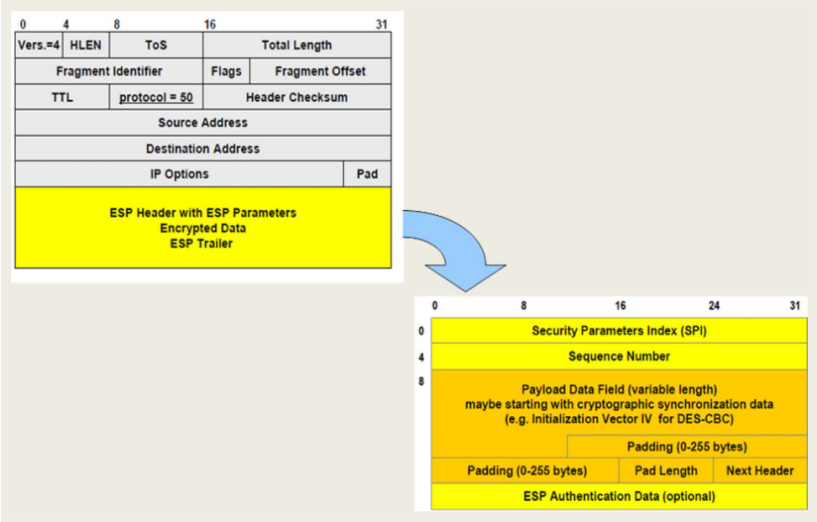


Figure 11.6: ESP mode

Chapter 12

Trusted Zone & Intel SGX

There are two hardware-based mechanism developed to support security, in particular for IoT and ICS systems.

12.1 TrustZone - Overall architure

TrustZone technology does not provide a fixed "one-size-fits-all" security solution, but an infrastructure foundations so that a SoC designers can choose from a range of components that can fulfil specific functions within the security environment.

The main security architecture **goal** is quite simple: enable the construction of a programmable environment to protect from attacks to *confidentiality* and *integrity* of almost any asset.

A platform with these characteristics can be used to build a wide set of security solutions which are not cost-effective with traditional methods.

Trustzone aims to partition hardware and software resources into two zones:

Secure world *for the security subsystem*
Normal world *for everything else*

3 Key-Features

- **Hardware logic** ensures a strong security perimeter between the two so that *Normal world* components cannot access *Secure world* ones. By placing sensitive resources in the *Secure world*, and by robust software on secure cores, we protect almost any asset against possible attacks
- **Extensions** in the processor cores to share a single physical core between the *Normal world* and the *Secure world* in a time-sliced fashion. This removes the need for a dedicated security core
- A security-aware **debug infrastructure** which can enable control over access to *Secure world* debug, without impairing debug visibility

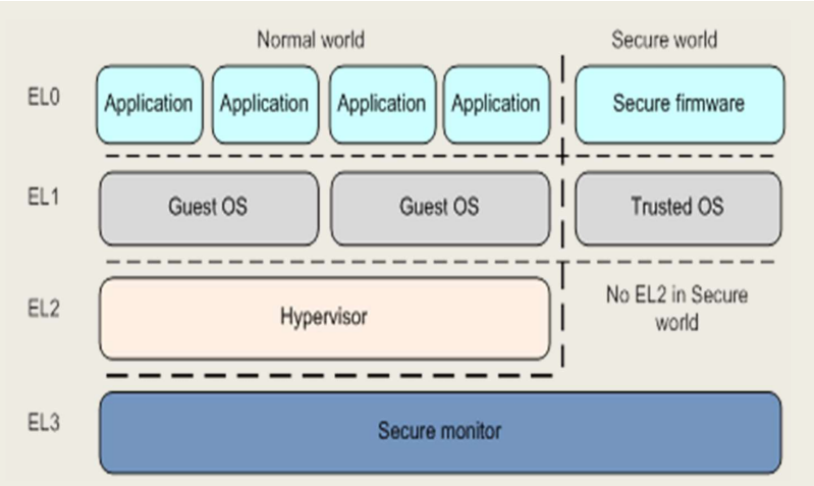


Figure 12.1: TrustZone overall architecture

12.2 Truszone - System architecture

12.2.1 System Bus

The most significant feature of the **extended bus** design is the addition of an **extra control signal**, the *Non-Secure* (or *NS*) bits for each of the read and write channels on the main system bus.

All bus masters set these signals on a new transaction, and the bus or slave decode logic interpret them to ensure separation is not violated.

All *Non-secure* masters **must** have their *NS bits* **set** in the hardware, which makes it impossible for them to access *Secure slaves*. In case they try, the address decode for the access will not match any *Secure slave* and the transaction will fail.

If a *Non-secure* master attempts to access a Secure slave it is implementation defined whether the operation fails silently or generates an error. An error may be raised by the slave or the bus, depending on the hardware peripheral design and bus configuration.

12.2.2 Processor

Each physical processor cores provides two **virtual cores** —one Non-secure and the other Secure— and a **robust context switch** between them, known as *monitor mode*.
The *NS* bit value sent on the main system bus is *indirectly derived* from the identity of the virtual core that executes the instruction or data access. This enables trivial integration of the virtual processors into the system security mechanism; the Non-secure virtual *processor* can only access Non-secure system *resources*, the while Secure virtual processor can see all resources.

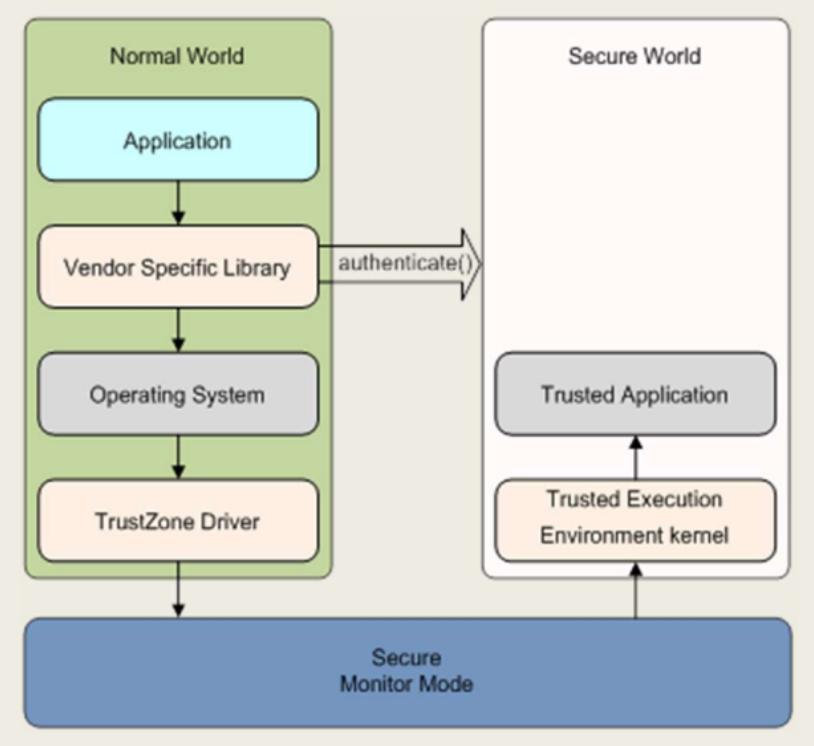


Figure 12.2: TrustZone processor architecture

Virtual Processor Switch

The two **virtual** processors execute instructions in a **time-sliced** fashion, context switching through a new core mode called monitor mode when changing the currently running virtual processor.

The mechanisms the physical processor uses to enter monitor mode from the Normal world are tightly controlled, and are all viewed as *exceptions* to the monitor mode software.

The entry to monitor is triggered by a dedicated instruction, the *Secure Monitor Call* (SMC) instruction, or by a subset of the hardware exception mechanisms. Interrupts and exceptions can all be configured to cause the processor to switch into monitor mode.

The software that executes within monitor mode is implementation-dependant, but it generally saves the **state** of the current world and restores the state of the world being switched to. It then performs a return-from-exception to restart processing in the restored world.

The world where the processor is executing is indicated by the NS-bit in the *Secure Configuration Register* (SCR) in CP15, the system control coprocessor, unless the processor is in monitor mode; since in that case, the processor is always executing in the Secure world regardless of the value of the *SCR NS-bit*, but operations will access Normal world copies if the *SCR NS-bit* is set to 1.

12.2.3 Monitor

The **monitor mode** software provides a robust **gatekeeper** which manages the **switches** between the Secure and Non-secure processor states.

Its functionality are similar to a traditional *OS context switch*, ensuring that **state** of the world that the processor is leaving is *safely saved*, and the state of the world the processor is switching to is *correctly restored*.

Normal world **entry** to monitor mode is tightly **controlled**. It is only possible via the followings: an *interrupt*, an *external abort*, or an *explicit call* via an SMC instruction.

The Secure world **entry** to the monitor mode is a little *more flexible*, and can be achieved by **directly writing** to **CPSR** in addition to the exception mechanisms available to the Normal world.

The **monitor** is a security **critical component**, as it provides the interface between the two worlds. For robustness reasons it is suggested that the monitor code executes with *interrupts* ***disabled***.

12.2.4 Memory subsystem

Two virtual **MMUs**[1] exist, one for each virtual processor. Each world has local set of translation tables, giving **independent control** over virtual to physical **mappings**.

The L1 translation table descriptor includes an *NS field* the Secure virtual processor uses to determine the value of the NS-bit to access the physical memory locations associated with that table descriptor.

The Non-secure virtual processor hardware ignores this field and $NS = 1$ in any memory access. This enables the Secure virtual processor to *access either* Secure or Non-secure memory.

To enable efficient context switching between worlds, **entries** in the *Translation Lookaside Buffer*s (TLBs) are **tagged** with the **identity** of the world that performed the walk. Non-secure and Secure entries *co-exist* in the TLBs, enabling faster switching *avoiding* the need to flush TLB entries for each context switch.

To enable this the L1 —and where applicable L2 and beyond— caches have been **extended** with an additional **tag bit** to record the security state of the transaction that accessed the memory.

The cache content with regard to the security state is dynamic. Any non-locked down cache line can be **evicted regardless** of its *security state*. A Secure line load may evict a Non-secure line and a Non- secure load may evict a Secure line.

12.2.5 Interrupts

Two interrupt lines exist, **IRQ** and **FIQ**, trapped in the monitor, without intervention of code in either world.

Once the execution reaches the monitor, the trusted software routes the interrupt request accordingly. This allows a design to provide **secure interrupt sources** the Normal world software *cannot manipulate*.

The recommended model uses *IRQ* as a Normal world interrupt source, and *FIQ* as the Secure world source. *IRQ* is the most common interrupt source in most operating environments, so the use of *FIQ* as the secure interrupt should mean the fewest modifications to existing software.

If the processor is running the **correct** virtual core when an interrupt occurs there is **no switch** to the monitor and the interrupt is handled locally in the current world. Otherwise the hardware traps to the monitor that causes a *context switch* and jumps to the restored world, at which point the interrupt is taken.

[1] *Memory Management Unit*

12.2.6 Debug

The debug extensions separate the debug access control into independently configurable views of each of the following aspects:

- ◇ Secure privileged invasive debug
- ◇ Secure privileged non-invasive debug
- ◇ Secure user invasive debug
- ◇ Secure user non-invasive debug

The Secure user mode debug access is controlled by two bits, SUIDEN (invasive) and SUNIDEN (non-invasive) in a Secure privileged access only CP15 register. This enable a processor to give control over the debug visibility once the device is deployed. It is possible to give full Normal world debug visibility while also preventing all Secure world debug.

12.2.7 Secure OS

A **secure OS** can simulate **concurrent execution** of multiple Secure world applications, run-time download of new security applications, and Secure world tasks, **completely independently** from the Normal world environment.

An extreme version of these designs closely resembles the software stacks in a *Soc* with two physical processors in an Asymmetric Multi-Processor.

Each virtual processor runs a standalone operating system, and each world uses hardware interrupts to preempt the currently running world and acquire processor time.

A tightly integrated design may uses a communications protocol that associates Secure world tasks with the Normal world thread that requested them. This provides many benefits of a Symmetric Multi-Processing (SMP).

In these designs a Secure world application could, for example, inherit the priority of the Normal world task that it is assisting. This would enable some form of soft real-time response for media applications.

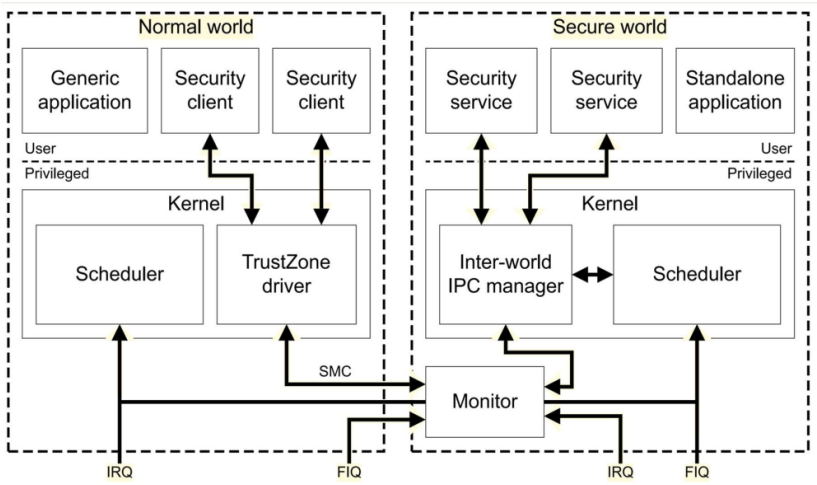


Figure 12.3: TrustZone secureOS

12.3 Intel Software Guard Extensions SGX

12.3.1 Enclave

SGX introduces notion of enclave Hypervisor:

- ◇ Isolated memory region for code and data
- ◇ New CPU instructions to manipulate enclaves and new enclave execution mode

Enclave memory encrypted and integrity-protected by hardware

- ◇ Memory encryption engine (MEE)
- ◇ No plaintext secrets in main memory

Enclave memory can be accessed only by enclave code

- ◇ Protection from privileged code (OS, hypervisor)

Application has ability to defend secrets

- ◇ Attack surface reduced to just enclaves and CPU
- ◇ Compromised software cannot steal application secrets

12.3.2 Construction

12.3.3 Measurement

12.3.4 Attestation

Chapter 13

Intrusion Detection

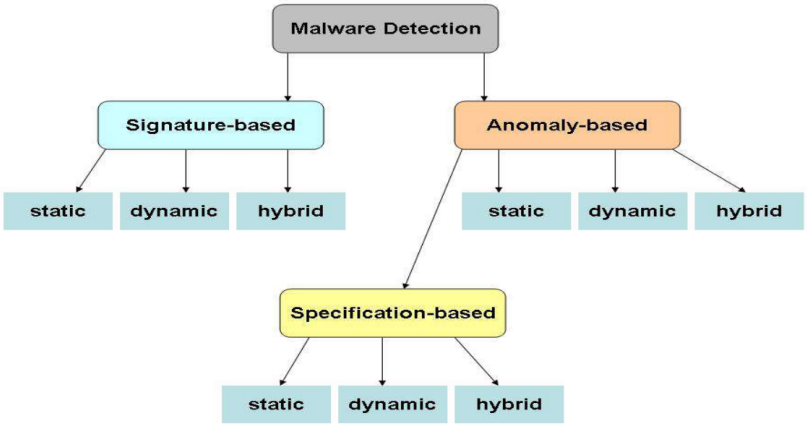


Figure 13.1: Intrusion Detection Taxonomy

INPUT-EVENTS

1. End-point events
   i. Invocation to OS
   ii. Memory analysis
   iii. Files that are downloaded
   iv. Programs that are executed
2. Network events
   i. Contents of packets that are transmitted
   ii. Information transmitted on a circuit

Networks events are generated by a module that monitors ongoing communication, i.e. a **packer sniffer**.
This poses two problems:

1. *Lost messages*, since the sniffer cannot slow down the communication it is sniffing
2. *Assumptions* on the *behavior* of the receiving node

There are ***evasion*** *attacks* where the attacker transmits **fake packets** to increase the computational load od the detection, exploiting fake checksum, overlapping fragments etc.

13.1 Anomaly Based

The behavior of the target system is observed for a time interval and a **learning model** is built representing the *normal* behavior.
Note that **learning** implies discovering parameters such as:

- ◇ **Services** that are used and time of the usage
- ◇ When users **log in** and the length of their **sessions**
- ◇ User **requests** and **OS functions** they invoke
- ◇ Computation and communication **bandwidths** used

After the learning phase, any behavior that is too *far* from the model that has been built is defined as an **anomaly** due to an *ongoing intrusion.*Clearly the critical parameters are the amount of **information** acquired during the learning phase and the **threshold** on the *"distance"*. Sometimes continuous learning is preferred since the normal behavior **changes** as time passes.

1. **Dynamic**: Information on a program behavior are collected by executing the program
2. **Static**: A static analysis returns information on the program behavior
   e.g. the OS functions it calls, information on the call order, etc.
3. **Hybrid**: Dynamic collection of information to cover lack of information in the output of the static analysis

13.1.1 Specification Based

The key point here is that the so-called normal network behaviour is not deduced by observing programs behaviour over time, instead by what are the running programs and what they should do according to their specification.

13.2 Signature Based

The main idea is that there are some **behaviors** and some **data** in a *malware* that **identify** the malware in a reliable way, i.e. *malware signature.*
All the signatures are stored in a database that is used to discover malware, hence two issues arise:

1. How to discover a signature
2. How to update the database

Note that this kind of detection needs malware signatures, thus it cannot detect an attack exploiting a 0-day vulnerability.

A 0-day exploit can be discovered through anomaly detection or by analyzing the information that a —for-intelligence— honeypot returns, however thare also alternative strategies exist to define a signature, for instance extending the approach by uploading suspicious code on the system of the tool supplier when a partial matching occurs. This is a solution inspired by *default allow*, i.e. anything that does not match the signature is allowed.

Signature based detection methods can be classified as follows

1. **Dynamic**: The program runs in a protected environment (virtual machine or sandbox), and then the collected information is compared against the signature
2. **Static**: The program code is analyzed, and the output is compared against the signature.
   This was the standard approach in old antivirus tools, which nowdays instead apply a hybrid one
3. **Hybrid**: Merge of the two previous approaches: a static analysis collects suspicious programs and then the behavior of each program is monitored.

A simple antivirus scanner performing **static** analysis matches code fragments against signatures, uses heuristic strategies to discover viruses and polymorphic viruses, and performs integrity checks on files to discover malicious updates by pairing files with checksum, hash, keyed hash etc.

Dynamic solutions typically use a *decryption and emulation* approach: The suspicious code is uploaded on a remote system (cloud run by the antivirus supplier) which emulates the execution for a predefined number of instructions. If the executions yields the expected results, the code is classified as safe. Clearly this does not discover viruses hidden within normal code.

Chapter 14

Polymorphic malwares and Sandboxes

14.1 Polymorphic malwares and viruses

Even if attackers are lazy, they react to countermeasures, and in fact have developed mechanisms to hide (obscure) information in the program to prevent a positive match against signatures.

14.1.1 Encryption

A tecnique is to encrypt malware and viruses. Upon infection, each copy of the virus program creates a new version by generating a new key and by encrypting the body of the virus. Sometimes the decryptor code is prepended to the actual malware, but even if not, it must exist somewhere and it can be detected.

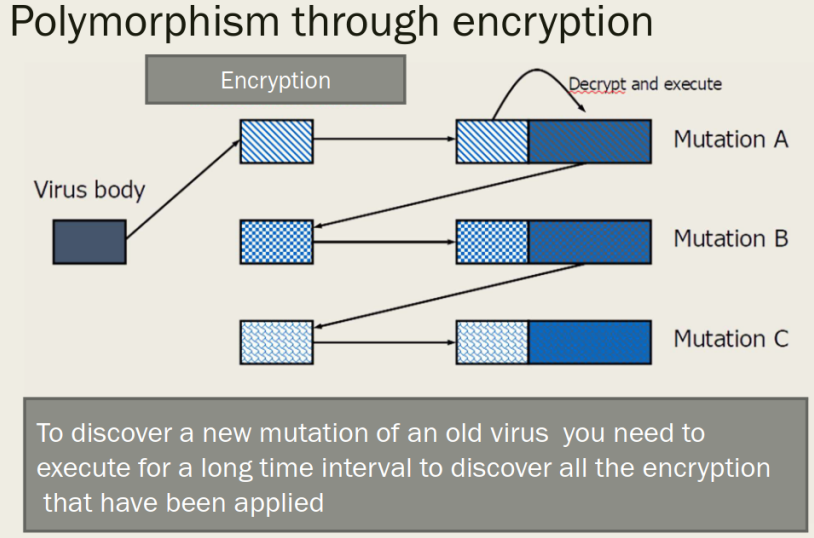


Figure 14.1: Polymorphism through encryption

14.1.2 `Emotet` example

A new Emotet wave was observed in late January 2022.13 that introduced the use of the `mshta.exe` application to carry out the infection, which is a legal Windows-native utility which *Microsoft HTML Application* (`HTA`) files "fooled" into performing malicious actions[1].

HTA files are basically HTM with enhanced privileges. When encoding, HTA allows the developer to have the features of HTML together with the advantages of scripting languages that sometimes are not present for html-based.

Emotet provided malicious HTA files containing highly obfuscated JScript code. On a dataset with 19,791 samples with non-trivial execution chains, 139 unique program chains and 20,955 unique invocation chains were identified.

[1]This kind of attack is known as *confused deputy attack*

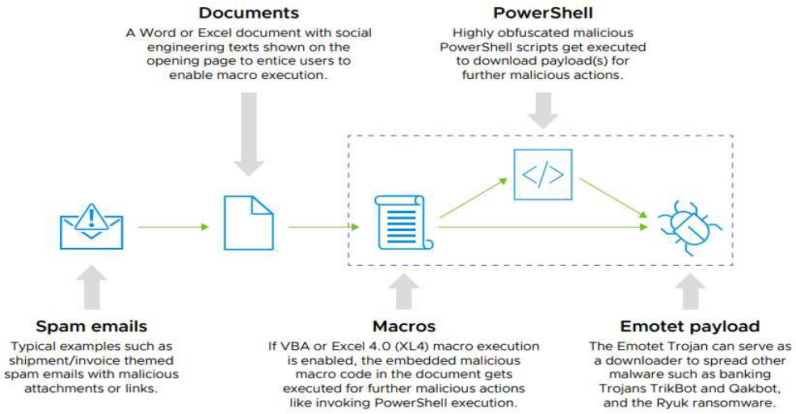


Figure 14.2: Emotet

14.1.3 `Zmist` example

Designed in 2001 by the russian virus writer `Z0mbie` The virus fragments are interleaved with fragments of the host application and are stored at random positions connected by memory jumps. When and if the virus is executed, it will infect any executable. The starting point of the virus cannot be reached in some execution because it is randomly generated.

14.2 Sandboxes

Recall that a **sandbox** is a *protected environment* to download and execute potentially risky code; it is disjoint and confined from the normal execution environment and usually hosted on a cloud. It allows to analyze and test a code without interaction with the system to be protected. Ideally, in a sandbox the code can do anything but escaping and thus accessing resources of the real system.

A sandbox is nothing but a specific purpose, highly robust **virtual machine** that detects anomalous behavior

(erasing/ encrypting files etc...)

Sandboxes can be used both for defence and analysis. They can be **riskful**, since if the malware escapes the sandbox then it may cause damage to the system. Besides nowdays malwares can discover whether they are running in a VM and defeat the detection by behaving correctly in the VM.

14.2.1 Detecting Sandboxes

Most VMs require that the VM runs some software tools, called guest additions which support file sharing between the host and the VM or even simple copy and paste operations between applications on the host and the VM. The presence of such guest additions is one of the easiest and most direct things to check to detect a VM.

To communicate from inside the VM to the host and vice versa, VMMs use things like shared memory or special instruction sequences. Even if the guest additions are not installed, these mechanisms are there and can be detected.

Besides, some malware look for signs of a system used by a normal user doing routine things as opposed to a clean system for a special purpose, like analysing malware. Usually, malware analysis starts with a *clean VM* because it is simpler to get a clean VM going for each malware analysis and having a clean system removes a lot of variability.

There is a debate on compatibility against transparency.

14.2.2 More-specific detection

Digging in deeper into VMs detection, we can consider a few known ways for malwares to understand whether they are running inside a VM environment.

- ◇ Assembly implementation of the `IsDebuggerPresent` API function: can indicate whether the current process is being debugged by checking the `BeingDebugged` flag of the *Process Environment Block*. The malware can execute a `CPUID` instruction with `EAX=40000000` and check if the returned value contains the string `"vmware"`.
- ◇ `NtGlobalFlag` field anti-debugging: indicates if the process was created by the debugger

- ◇ `RDTSC` instruction: indicates how many CPU ticks have taken place since the processor was reset
- ◇ *Stack Segment Register*: used to detect if the program is being traced.
- ◇ Malware usually uses the `Sleep` API (`Beep` API in *Windows*) function to delay execution and avoid detection by sandboxes.

14.2.3 `WannaCry` example

`WannaCry` was malware first appeared back in 2017. It was a worm which encrypted the boot block. The malware tried to contact a non-existing website, if no response was given it would have infected other nodes, otherwise it would have not. Sandboxes (typically) always answer positively to attempts to contact outside network, making the malware perfectly designed to determine whether it was running in a sandbox or not, since in a real environment no response would have been received, since the website doesn't exist.

Chapter 15

Detection Tools

15.1 Rule based Signature Detection

A strategy to generalize the notion of signature is the matching of a *set of **rules***: A set of rules is more absract than a regular expression or a string, and such generalization can tolerate some changes in the body of the malware.

Yara rules are becoming the de facto standard for pattern matching, which can be applied in many contexts, including IDS rules but not only; it has been made to discover and identify malware samples.

A well-known rule based detection tool is **Snort** IDS[1]:

1. A good **sniffer**
2. A **rule** based detection engine
3. Packets that do not match rules are neglected (by the analysis)
4. Packet that match rules are analyzed and can even fire an advice

15.2 Yara

Yara is quite simple:

1. Open a text editor
2. Write rules in Yara syntax
3. Give Yara a set of files to be analyze
4. Let Yara find evil for you

A tool to identify and classify malware samples by creating descriptions of malware families based on textual or binary patterns.

An example of a Yara rule is provided below:

```
rule silent_banker : banker
{  meta:
      description = "This is just an example"
      threat_level = 3
      In_the_wild = true
   strings:
      $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
      $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59    F7 F9}
      $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
   condition:
      $a or $b or $c
}
```

In this example any file/memory area containing one of the three `strings` must be reported as `silent_banker`.

[1] *Intrusion Detection System*

Cuckoo Module

There are also Yara modules used to introduce other features aside from Yara's native ones.

A relevant example is the `Cuckoo` module which allows to run a piece of code, capture some information about the behaviour of the code and pass it to Yara, making the performed analysis dynamic.

```
import "cuckoo"
rule evil_doer
{
   string os:
      $some_string = { 01 02 03 00 05 06 }

   condition:
      $some string and
      cuckoo.netuork.http request(/http: vvsomeooe,eoioge 1 cam')
 }
```

15.3 Snort

Snort, based on `libpcap`, provides three possible modes to operate in:

1. **Sniffer** To output data in transit and also check the IP and TCP/ICMP/UDP headers: `./snort -vd`
2. **Logging** In case wanto to designate a logging directory . `./snort -vd`
3. **NIDS**[2]: NIDS mode allows to define through rules the action to take upon detection of malicious data packets.

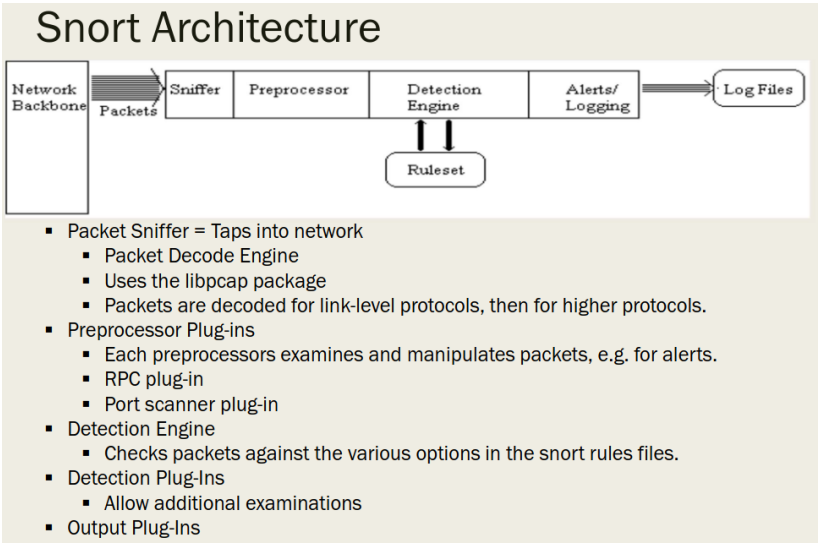


Figure 15.1: Snort architecture

[2] ***Network*** *Intrusion Detection System*

Port Scanning - plugin

Two common practices made by attackers to scan a network are port scanning, sweeping and walking. There is a dedicated snort plugin to detect such actions.

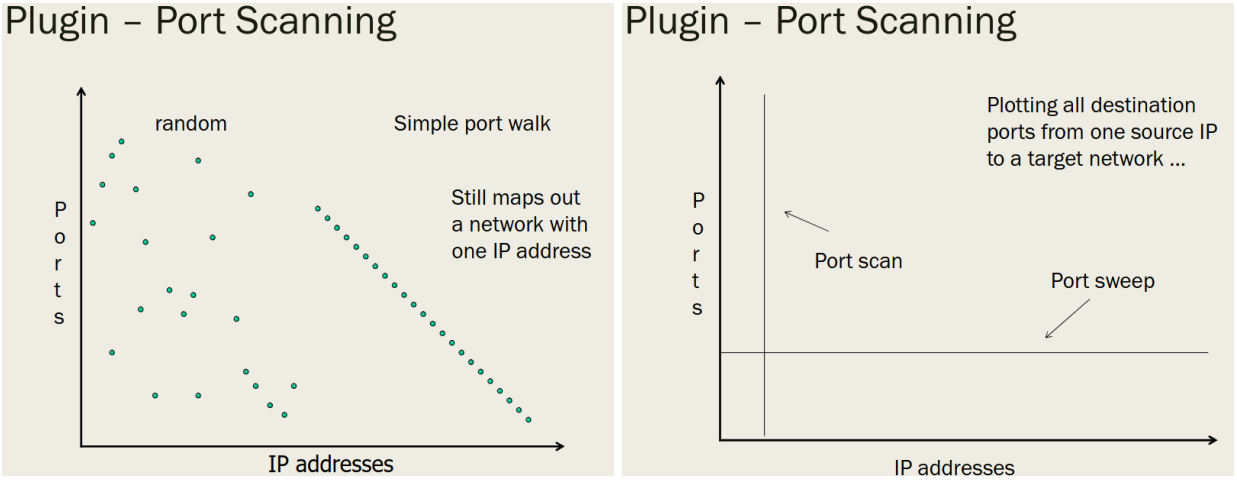


Figure 15.2: Snort port scanning plugin

15.3.1 Rules

Snort rules structure

1. **Rule header**
   i. The rule action,
   ii. The protocol
   iii. The source and destination IP addresses
   iv. Network mask
   v. Information on source and destination port
2. **Options section**
   i. Messages
   ii. Information on the packet sections to be examined to discover if the rule should be fired.

```
alert tcp any any -> any any (msg:"Possible Zeus Botnet C&C Traffic"; flow:established,
    to_server; content:"|5a 4f 4f 4d 00 00|"; depth:6; sid:1000005; rev:1;)
```

*Snort predefined **actions***

1. `alert`: raise an alert and log the packet
2. `log`: log the packet
3. `pass`: neglect the packet
4. `activate`: raise an alert and activate a dynamic rule (remember a state)
5. `dynamic`: idle till it is activated then logs the packets
6. `drop`: log and block the packet (not only a sniffer but also a filter)
7. `reject`: block and log the packet and then send
   i. TCP reset if TCP
   ii. ICMP port unreacheable if UDP
8. `sdrop`: block the packet but do not log it

Multiple rules might match a packet, thus an order must be established. The safest way is to apply rules in the order

```
Drop > Pass > Alert > Log
```

This order is safe yet very expensive in terms of computational resources and time, since the `DROP` check *overhead* is computed even for legal packets. Since, most packets are legal and not dangerous, a more effective but more dangerous option is

```
Pass > Drop > Alert > Log
```

15.4 Merging Signatures and Anomalies

15.4.1 Endpoint Detection and Response

An **EDR** or an *endpoint threat detection and response* (ETDR) is an endpoint security solution that merges monitoring and collecting information in real time: *Real-time* **collected** information is analized and proper responses are fired by a **rule**-based system.

EDR/ETDR is used to describe systems currently developed that automate the analysis of information and the responses to detected intrusions.

The **main tasks** of an ETDR include:

- *Endpoint monitoring* to collect information about possible intrusions
- *Analyze and correlate* collected information to discover ongoing intrusions
- *Automatic response* to intrusion to minimize the impacts
- Apply forensics tool to discover intrusions
  (even old ones)

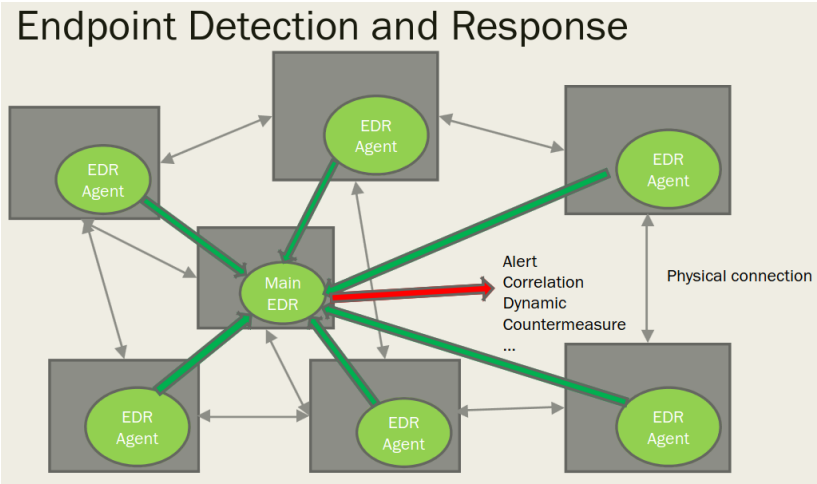


Figure 15.3: EDTR architecture

Main Components

- **Endpoint data collection agents**: collects information on security status and transmits it to a data *collection center*.
  It can also log information, apply patches, activate or kill processes.
  It may analyze the memory, the files or the packets flowing across the node or implement anomaly detection.
- **Centralized analytical engine**. It looks for patterns in the *real-time data* and signals if an intrusion involves one or several nodes. It also coordinates the actions of the various agents.
  It may use IOC, signature etc.
- **Forensics analysis**: An EDR may implenent forensic analysis to discover intrusion that have exploited new vulnerabilities and new attack techniques. The forensics component may discover old undetected intrusions to produce data on still unknown techniques. This analysis does not need to be run in real time.

Looks like gold, but in fact, there is **not** much **correlation** happening in the central node, and EDR agents are not so powerful defenders, even if more advanced than legacy antiviruses, since they monitor the ongoing operation, scan images as they are run or the node memory, covering a much larger number of attacks.

The main **goal** of **EDR** is *automating* the work of a defender. This reduces the investment in cyber security and enables an organization to *monitor* its infrastructure.

15.4.2 Introspection

A mechanism which resides in the EDR family, is **Virtual Machine Introspection** which formally defines techniques and tools to monitor a VM run time behavior to protect a VM from internal and external attacks

Key-Points

- ◇ Along with introspection some basic security mechanism are provided, such as *virus scanners* and *IDS*s.
- ◇ Observe and respond to VM events from a *"safe"* location *outside* the monitored machine.
- ◇ Exploit virtualization as a countermeasure. It is an example of how virtualization changes the computing framework.
- ◇ With respect to static attestation, VMI implements a form of run time attestation that aims to discover not only which software a system runs but also its run time integrity.

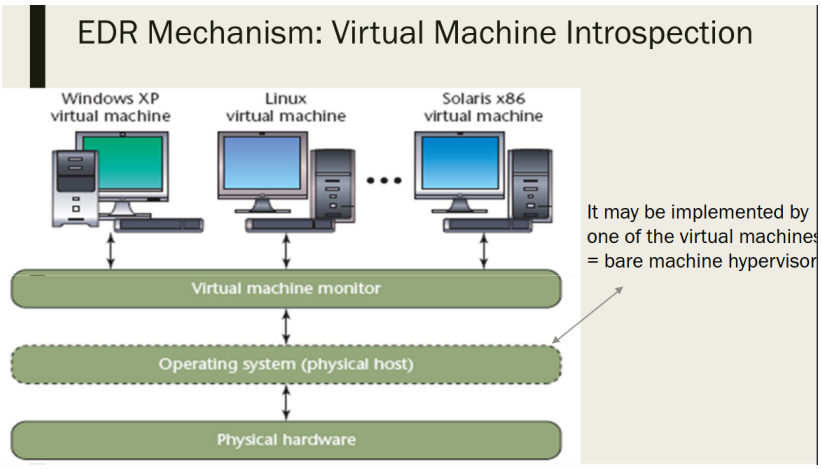


Figure 15.4: VMI arch

VMI defines a **Memory Mapping** system[3] to seamlessly and easily manage memory accesses, through advanced memory translation.

Besides there is a mechanism to check the integrity of virtual components, based on a chain of trust, which is both a strength and a possible point-of-failure.

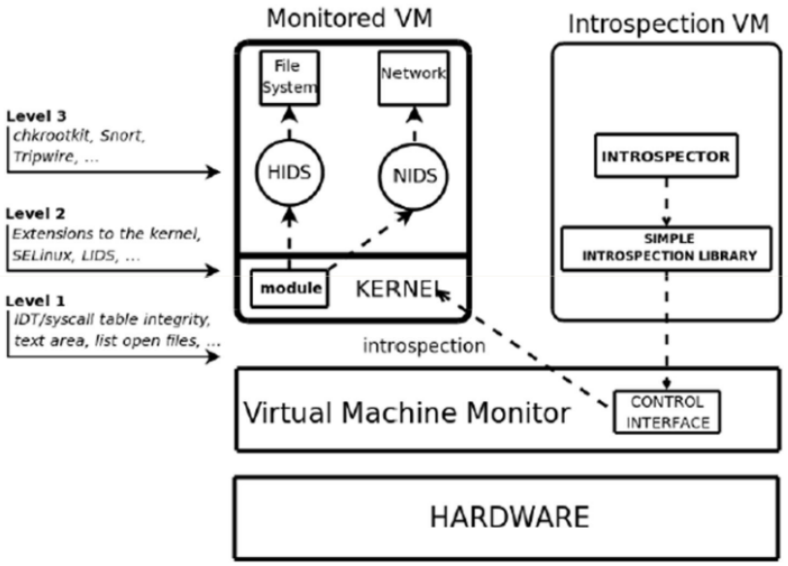


Figure 15.5: VMI Chain of Trust

Chain of trust - Intuitive

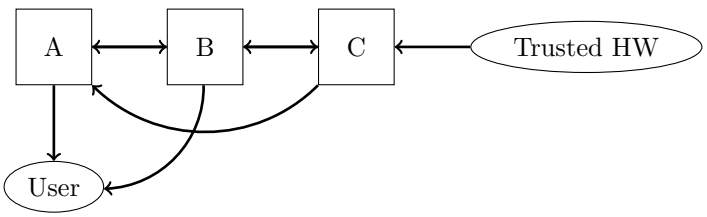


Figure 15.6: Chain of trust simplified schema

[3]Actually it is not clear from the slides where does this Memory mapping comes out

Considering Fig 15.6 I want to be sure that A runs the code I want, so B computes the hash of the code and shows it to me. I trust B because C computes the hash code and shows it to me. Clearly, we can endlessly chain multiple checks, but such chain is valid only if in the root there is *hardware* which can be *trusted*.

Chapter 16

Intrusion Analysis

Discover possible intrusions A critical point in improving the robustness of a system and resist to intrusions is to know all the intrusions of one or some threat agents aka attackers; the reason is that we are sure that our changes improve the robustness and resilience of a system only if we know all the intrusions.

If we know just a ***proper subset*** *Sub of the possible intrusions Int against a system S and if we change S to stop intrusion in Sub only, then this may increase the success probability of some intrusions against S in Int − Sub*

There is also a formal proof of the above *theorem*, which is a consequence of **Bayes theorem** and shows the holistic nature of security, meaning that you **cannot** achieve security by **local** improvements.

The theorem can explain the failure of risk assessment and management solutions based upon the discovery of a few intrusions against a system (penetration test, red/purple team, capture the flag) and stresses that automated discovery is needed due to the huge number of intrusions.

16.1 Bayes theorem

Bayes Theorem came out from logistic and operational research and states the following: *Let us assume there is a group of people that want to go from A to B but all the paths between the two points cross a bridge, then an increase in the number of bridges may increase the average time from A to B.*

In *cybersecurity* the increase may be due to lack of information on the **shortest paths** from A to B. By increasing the number of paths we increase uncertainty about the optimal choice at a choke point. Looking at the theorem from an opposite point of view, a decrease in the number of possible intrusions (= *paths*) may decrease the time to implement an intrusion.

16.2 Measuring #intrusions

16.2.1 Graph

Build an oriented graph OG paired with the triple $\langle system\ S, attacker\ TA, goal\ G\rangle$ as following:

1. Each node N of OG is paired with a set of access rights of $S = security\ status = SS(N)$ =access rights TA has acquired through the previous attacks
2. Each arc A of OG is labelled with $\langle A, V, M\rangle$ where A is an attack, V a vulnerability and M is a module of S
3. The *precondition* of A (i.e. the access rights to execute A) are a subset of $SS(N)$ where N is the source of A

Graph properties

- ◇ OG is acyclic (TA is rational and minimizes its efforts)
- ◇ No path of OG has multiple occurrences of an arc with the same label
- ◇ The initial node I is the one where $SS(I)$ is the set of legal access rights of TA
- ◇ A node FN is final if there is at least one path from I to FN and the FN is the first node on the path where $SS(FN)$ includes G
- ◇ Any intrusion defines a path from I to a final node
- ◇ G can be generalized to a set of goals and node N is final if $SS(N)$ belongs to G

Note that attack graphs assume that attacks *always succeed*, and that access rights are *never lost*, in fact they are always

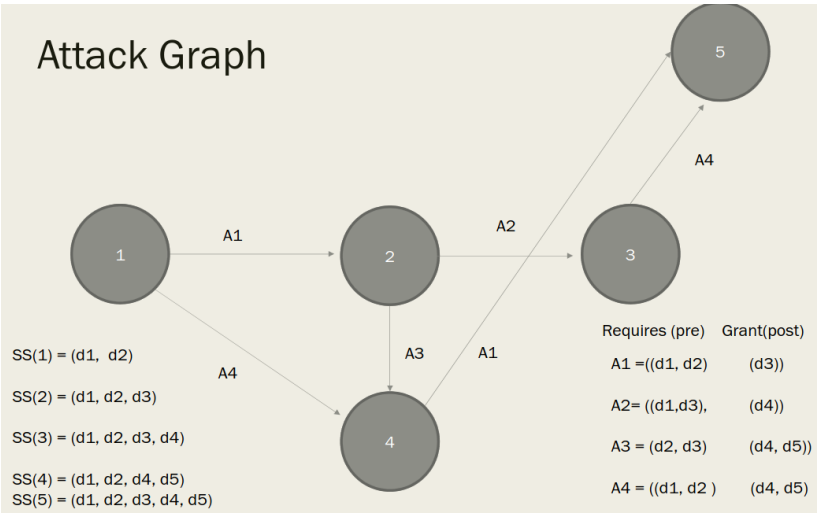


Figure 16.1: Simple attack graph example

increasing

Since no defense is considered the acquisition of rights is **monotonically increasing**, leading to **huge graphs**, making them computationally unfeasable to be analyzed. To consider attack failures we need to define an **attacker state** that also takes into account previous **failures**, since overall process is not a Markov process because the next action also depends upon the past history. However, a detailed modeling of attacks and access rights increases the overall complexity.

16.2.2 Tree

We can simplify the attack graph and represent intrusions with trees:

1. A leaf represents an attack enabled by a vulnerability = an elementary attack
2. a node that is not a leaf represents a complex attack
3. the subtree rooted in the node shows how a complex attack may be implemented

- ◇ Each node that cannnot be mapped into an elementary attack should be then decomposed into a sequence of attack
- ◇ The leaves of the tree represents the sequence of attacks in an intrusion
- ◇ We can have an AND/OR trees where a not leaf node may have AND or OR sons:
  - AND = All the attacks that are sons of the node have to be executed
  - OR = One attack sufficies
  - The son of AND/OR nodes may be either elementary or complex attacks
- ◇ Useful to represent or discover a decomposition and possible choices but not to discover all intrusions

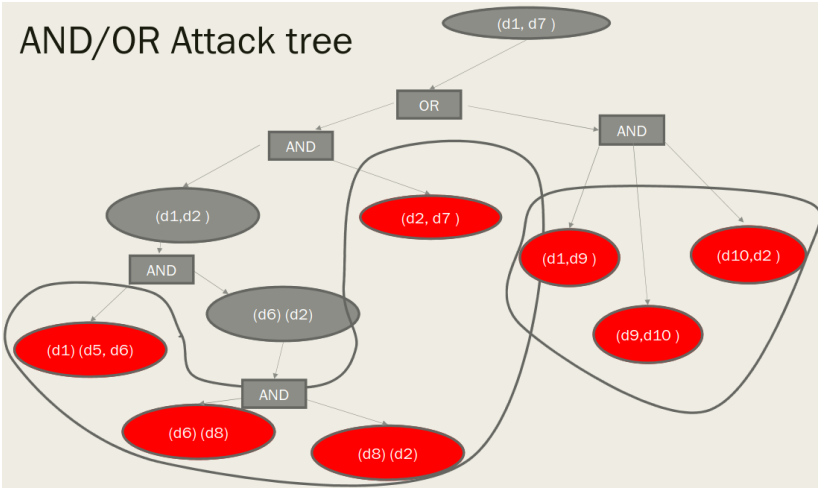


Figure 16.2: Attack tree

Spoiler alert: intrusion-trees do not work either ☺

16.2.3 Building and stopping intrusions

Building intrusions

The critical point to discover intrusions is **adversary emulation**, understanding how attackers behaves and how they can chain actions to reach a goal.

A starting point may be given by the following properties of an adversary

1. Attacks they may execute (due to preferences, tools, resources, know how)
2. Initial access rights = the legal ones
3. Initial informations on the system
4. Final access rights = those in a goal

Recall that an attacker **cannot** build an attack graph before starting it intrusion since they have limited visibility and thus *lack information* on system components. The attacker interleaves the building of a map of the whole system with the attacks to reach its goal, possibly resulting in the execution of *"useless" attacks* that do not grant any access right to reach the goal, but are try-and-error steps towards defining such map.

Insiders are among the most dangerous attackers because they do not have to collect information as they already have a map

Stopping an intrusion

To stop an attacker we have to stop all its intrusions, and to stop an intrusion we need to stop at least one attack in the attack chain in an intrusion (stopping other actions may be more difficult).

It is important to focus only to impactful attacks, the so-called "useless" ones are negligible in the analysis. To **optimize** the security investment, we should aim to stop attacks useful for several intrusions: choose the *minimal* number of attacks to stop all intrusions or choose a set of attacks to stop such that there is at least *one attack for each intrusion* and the overall cost is the lowest one.

This optimization problem highlights how to rank vulnerability to produce a ranking tailored for the target system: the score of a vulnerability v increases with the number of chains or paths we can stop by removing the vulnerability v. In the attack graph, tailor the score to the pair $\langle system, attacker \rangle$ by considering the paths the attacker can implement.

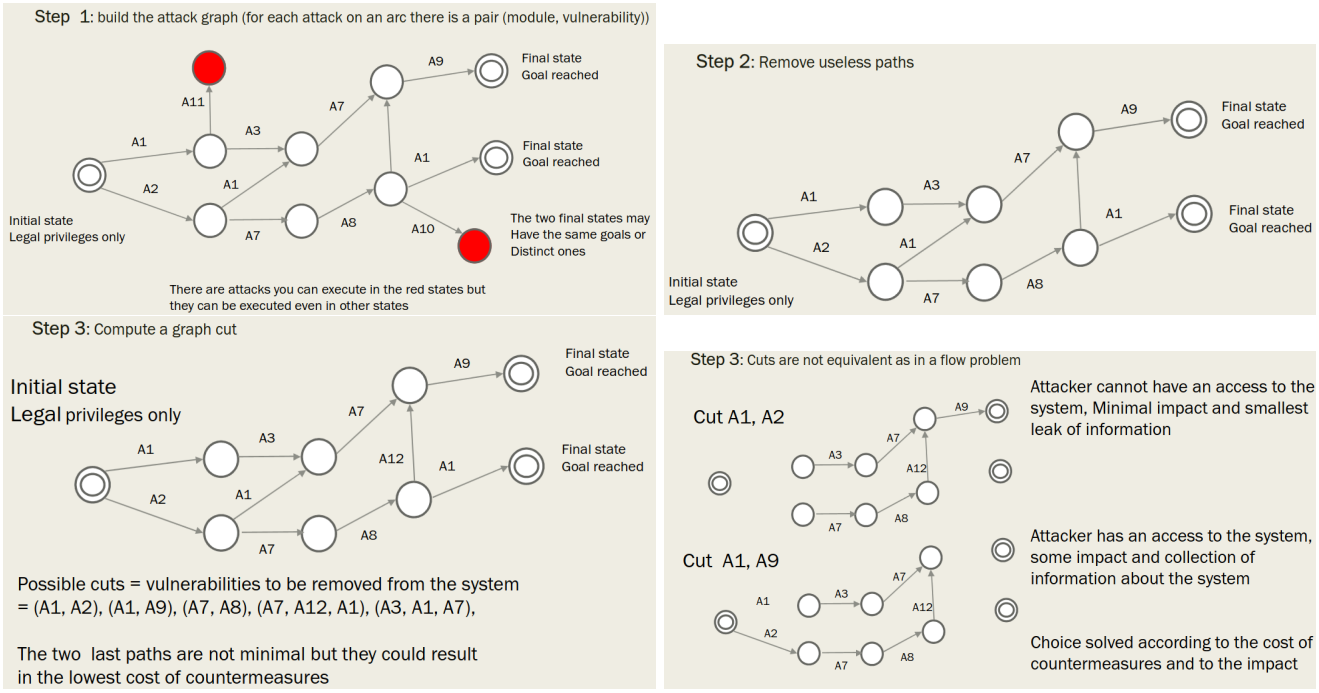


Figure 16.3: Stopping an intrusion using graphs

16.3 Automating intrusion

Even if it's possible to automate attacks, automating intrusions is more complex since doing so implies having a strategy on ordering attacks and alternating them with actions.

In simple cases where the ordering of actions can be easily discovered automation is possible, otherwise only some phases are automated but not the whole intrusion.

Intrusion steps

- ◇ Initial access to the system ⟵ *Automatable*
- ◇ Lateral movement and information collection
- ◇ Deployment of malware and encryption ⟵ *Automatable*

Instead of using the standard previously described attack graph, a more realistic solution is the following:

1. **Emulate** the behavior of the various attackers
2. Record each action in a graph
3. Merge the graphs
4. Analyze the graph describing their intrusions (a posteriori graph)
5. Compute a **cut** of the graph

An important feature of an **emulation** is the modeling of attack **failure** and how they are handled by an attacker, which can be done in various ways:

1. Repeat the attack till it succeeds
2. Repeat for a predefined number of times and then forget
3. Choose another action but do not forget the attack and come back later.
   The last strategy may result in the discovery of **black swans**, which are events with a very low but ***not*** *neglectable* probability.

16.3.1 Possible attacker's actions

Persistence

An attacker aiming to steal information or to attack later is interested in persisting in a system, i.e. acquiring permanent access to a system.

This can be achieved by, for instance, installing a backdoor or some trojan

Persistence may be also useful to restart an intrusion from an intermediate point when and if the intrusion has been detected. The modules the intrusion installs onto the system interact with the *command and control* infrastructure the attacker has created before starting the intrusion. To avoid easy detection and blacklisting, some attackers store the address of the nodes of the **C2**[1] Infrastructure in a public blockchain.

Evasion

This class of actions aims to defeat IDS and EDR tools. Some examples of mechanisms to evade a network signature-based IDS are

- ◇ Message fragmentation
- ◇ Message reconstruction
- ◇ Fake messages

Other examples, assessing different defendants countermeasures, are *token manipulation*, to evade control on authentication, or *locally building* container images on a host, evading controls on container images that are first retrieved and then installed on a host.

A frequent choice is to run malicious code inside a VM instance, allowing attackers to hide artifacts from security tools that are unable to monitor activity inside the virtual instance.

Notice that the evasion techniques the attacker applies changes the success probabilities of some attacks

16.3.2 Mitre att&ck matrix

Mitre att&ck matrix is —a public knowledge database— the de facto standard to describe possible behaviors of attackers, intended as the *steps* of an intrusion but not the intrusion *itself*.

[1] *Command & Control*

The matrix format slightly changes according to the type of attack discussed

1. Pre-Attack
2. Enterprise
3. Cloud
4. ICS
5. Mobile

Provides a hierarchical definition of behaviours:

1. **Tactics**: *–why–*
   short term goal in an intrusion
2. **Technques**: *–how–*
   how the short term goal is reached
3. **Procedure**: *–more detailed how–*
   detailed implementation of a technique

Enterprise (?) Tactics

1. **Reconnaissance**: gathering information to plan future intrusions
2. **Resource Development**: establishing resources to support operations ⟶ *setting up C2 infrastructure*
3. **Initial Access**: how to get into your network ⟶ *spear phishing*
4. **Execution**: run adversary-controlled code ⟶ *running a remote access tool*
5. **Persistence**: trying to maintain their foothold ⟶ *changing configurations*
6. **Privilege Escalation**: trying to gain higher-level permissions ⟶ *leveraging a vulnerability to higher access*
7. **Defense Evasion**: trying to avoid being detected ⟶ *using trusted processes to hide malware*
8. **Credential Access**: stealing accounts names and passwords ⟶ *keylogging*
9. **Discovery**: trying to figure out your environment ⟶ *exploring what they can control*
10. **Lateral Movement**: moving through the environment ⟶ *using legitimate credentials to pivot through multiple systems*
11. **Collection**: gathering data of interest to the adversary goal ⟶ *accessing data in cloud storage*
12. **Command and Control**: C2 communication with compromised systems ⟶ *mimicking normal web traffic to communicate with a victim network*
13. **Exfiltration**: stealing data ⟶ *transferring data to cloud account*
14. **Impact**: manipulate, interrupt, or destroy systems and data ⟶ *encrypting data with ransomware*

ICS

When considering attacks to ICSs, there are two other tactics that are *not* present in *Enterprise*:

1.
2. **Inhibit Response Function**: techniques that adversaries use to hinder the safeguards put in place for processes and products. They result in the inhibition of safety, protection, quality assurance, or operator intervention functions to disrupt safeguards that aim to prevent the loss of life, destruction of equipment, and disruption of production.
3. **Impair Process Control**: techniques to disrupt control logic and cause malicious effects to processes the target environment controls. Targets may include active procedures or parameters that manipulate the physical environment. These techniques can also include prevention or manipulation of reporting elements and control logic.

Exfiltration is missing because these systems are usually not attacked to steal information, while some techniques are typical of ICS, like manipulating an indicator on a host.

Cloud

Use of standard malicious software is not taken into account when considering attacks to cloud-based system, since they are easily detectable.

Instead, a typical way of attacking and exfiltration, is for attackers to have a cloud account and transfer information from other accounts to theirs, resulting in stolen information.

16.3.3 Exam Project Ideas

Looks for slides of 7^{th} *November*. Slides 10...14 provide a list of more specific examples of several tactics.

16.4 9 Novembre

Didn't take any notes today.

16.5 Caldera

Caldera is an open-source framework developed by MITRE for the creation of automated adversary emulation plans. It is designed to help security teams assess their organization's defences by simulating real-world attacks using a techniques and tactics mapped to the MITRE ATT&CK framework. This automated solution includes a graphical user interface (`GUI`) that allows users to easily create and manage custom attack plans, as well as a range of plugins and modules for different attack techniques. It shows an attack in real-time and allows security teams to identify their weaknesses in a controlled environment.

The simulation of an **adversary profile** corresponds to an operation structured into **phases**

- ◇ Each phase involves the execution of an ability, so there's a matrix with abilities and phases.
- ◇ Two abilities in different phases can communicate through facts the first one produces.
- ◇ A centralized architecture with one server and client-side agents, where each agent simulates an attacker on the executing host. The server administers the operations, plans the actions to be executed and returns to the user all the information from by the various agents.
- ◇ After planning actions to be performed, the server waits for the agents to connect and then order them what to do. Each agent executes on the host where it runs the action the server chooses.
- ◇ An agent can be started manually by the user or by another agent following some lateral movements in an operation

16.6 Cascade

Cascade is a command-and-control (C2) framework developed by MITRE for use in red team operations and adversary emulation; it is designed to enable red teams to quickly and easily create and manage C2 infrastructure, allowing them to more effectively simulate the actions of a real-world adversary.

Like *Caldera*, *Cascade* is based on the *MITRE ATT&CK* framework, and it is designed to support a wide range of different attack techniques and tactics by being highly modular. It can be integrated with other tools and platforms to provide a comprehensive adversary emulation solution for red teams and security practitioners.

Chapter 17

Reacting to Intrusions

Following an intrusion —theoretically— two things may happen:

1. Reaction on the **target system**, by performing incident response and management
2. Reaction on the **source of the intrusion**:
   The only useful action is to map the attack infrastructure to discover the systems the attacker uses to interact with the infrastructure, i.e. obtaining information useful to dismantle the infrastructure
   Any kind of counterattack on the source of intrusion is useless, we know the attacker is using an attack infrastructure

17.1 Reacting on the Target System

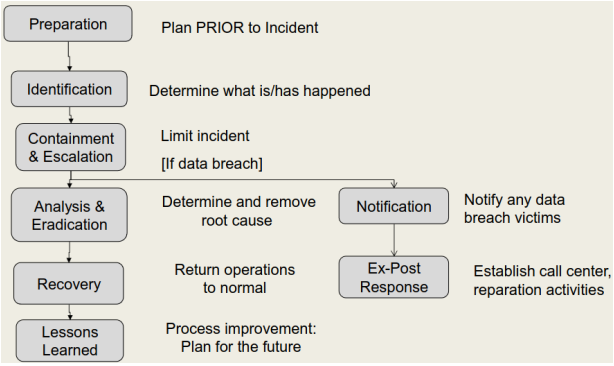


Figure 17.1: Reaction on the Target System

- **Containment**
  The main purpose of containment is to contain the damage and prevent further damage. The earlier intrusions are detected, the sooner they can be contained.
- **Eradication**
  Removes the threat and restoring affected systems to their previous state while minimizing data loss. It should remove malicious content and ensure the affected systems are completely clean but preserve evidence for later prosecution
- **Recovery**
  Testing, monitoring, and validating systems before putting them back into production to assure they are not re-infected or compromised (backup fundamentals, 3-2-1 rule)
- **Lessons Learned**
  It gives organizations the opportunity to update the target system to prevent future intrusion and to understand how to improve intrusion management.

17.1.1 Containment

Containment mechanisms

1. *Host quarantine*
   Host quarantine prevents an infected host from communicating with other hosts; it is implemented via IP-level access control lists on routers or firewalls.
2. *String-matching*
   String-matching containment matches network traffic against signatures of known worms to drop associated packets.
3. *Connection throttling.*
   Connection throttling proactively limits the rate of all outgoing connections made by a machine and thereby slowing —not stopping— worm spreading. It was proposed in a host context, but it can be applied to the network level as well.

17.1.2 Forensics

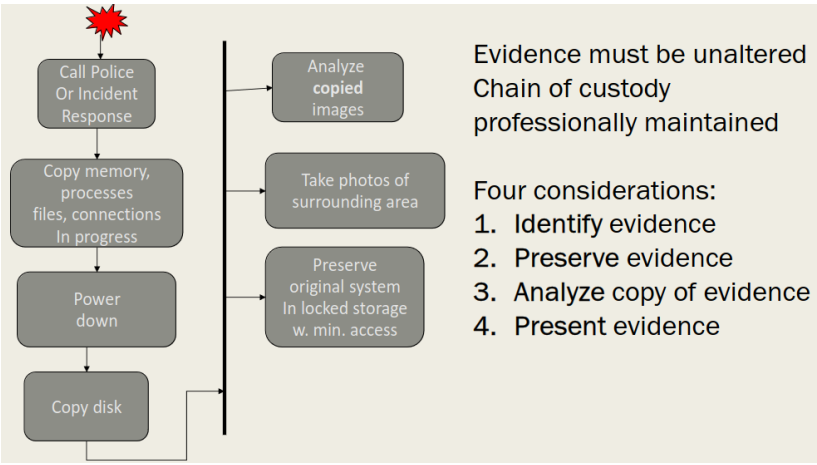


Figure 17.2: Intrusion forensics

Note that proper **logging** allows easier eradication and forensics analysis, where *"proper"* indicates a log file which records at least any of

- ◇ Login attempt
- ◇ Failed login
- ◇ Access to critical resources
- ◇ Anomalous messages (`Snort` log facility)

Besides, the log file should also be protected to guarantee its **integrity**, otherwise it's useless.

- ◇ Write once memory
  "paper-like"
- ◇ Insert a sequence number to discover log manipulation
- ◇ Insertion in a record of a value that is a function of all the previous records notehash pointer or blockchain

Lastly, to be useful from a **forensics** point of view, the log file should be structured so that it can be used to prosecute the attacker and be exploited as a **legal** source of evidence in an investigation and possibly chain of custody.

17.1.3 Compromise recording

Attackers who infect your system using *ransomwares* stole your data and then encrypt it on your local disks, then they ask for a ransom in order to have the data back. A few years ago, backups were very effective as a countermeasures to such attacks, and they still are a key point to achieve *resilience*[1], there is still the problem that the stolen data may contain sensitive or personal information, which should not be made public.

17.1.4 Security vs Debugging logging

When logging for **security** defending the **integrity** of the log from malicious manipulation is *fundamental*. If the integrity of a log for security is not assured, then the log is useless.

[1] i.e. put the system back on

If logging is performed for debugging — or more generally to improve performance, correctness, etc.— instead, integrity is not so crucial.

The difference between a sequence of blocks and a blockchain is a good example of the difference between collect for debugging and collect for security.

17.2 Attribution

Attribution, i.e. tracking and identifying the perpetrator of an intrusion results in a whole picture of the intrusions to enhance cybersecurity, besides, attribution can also improve the evaluation of the intrusion impact and increase the investment in countermeasures.

The information to attribute an intrusion are:

- ◇ Tools used
- ◇ Behaviors of the threat agent (to improve detection of future intrusions)
- ◇ The attack infrastructure
- ◇ Shared code
- ◇ The information collected in the intrusion (log + honeynet)

Attribution typically takes a long time and it may provide

- ◇ Information to eradicate the intrusion
- ◇ No immediate value to containing an intrusion
- ◇ No evidence for a court of law

Chapter 18

Automating Simple Intrusions

.There are several way to automate an intrusion that cover simple or sofisticated strategies, but currently there are no complete solutions for a targeted intrusion; only partial solutions which are handled by a human to convey a full intrusion.

One of the simplest strategy to automate an intrusion is a **worm** which spreads in Internet, attacks nodes and deploys some payload on each one, to allow further spreading.
It is very simple since it does not require an attack infrastructure [1]. Currently, worms one of the few cases that are fully automated.

18.1 Worms

A worm is a kind of malware that replicates itself onto other nodes without human intervention. There are several kinds of worms, depending on the way the worms spreads itself:

- ◇
- ◇ Email: hide in a message or in an attachment
- ◇ IRC
- ◇ Istant Messages
- ◇ File sharing: hide in file that a user download
- ◇ Internet or network worm: attack a node through a wormable vulnerability that affects the node

A vulnerability is *wormable* if it enables a remote attack that enables remote code execution; this results in the ability of creating a process running on the target node.

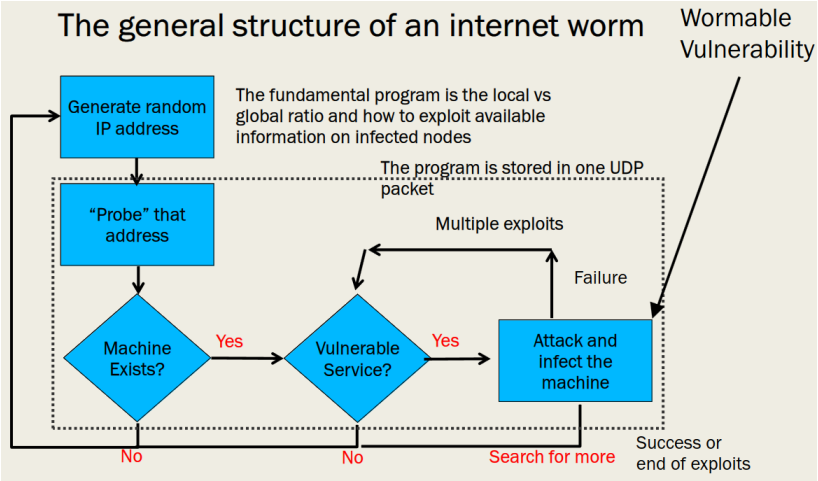


Figure 18.1: Worms structure

[1] sometimes it is used to build one instead

Obfuscation

In several worms the code is **obfuscated** to prevent antimalware software from detecting it.
Each subroutine in the code can be thought of as a state machine and implemented as a loop. At the start of each subroutine, the table of values is decrypted. This table of values serves as a container for constant values used in the subroutine, as well as the state transition table. The malware obfuscates calls to other subroutines. The subroutine address is computed using hard-coded values and values from the previously mentioned decrypted table of values. The result is placed in a register, and an indirect call is made using the register. A *fake* **payload** is uploaded if a sandbox is detected, while the *real* **payload** is a tor client.

18.1.1 Domain Flux and DGA

Most botnets use the DNS protocol to localize their C&C servers, and some use the **domain-flux** technique to evade blacklists.
Such technique implies generating a large set of pseudo-random domain names (`PDNs`) automatically. The infected machine may contact the bot-master using a *domain generation algorithm* (**DGA**) to produce a list of candidate C&C domains. Then the infected machine attempts to resolve these domain names by sending DNS queries until it obtains a successful answer from the malicious domain name reserved in advance by the bot-master.

This strategy is an *effective technique* to evade detection by a monitoring system. If one or more C&C domain names are identified and taken down, the bots will relocate C&C domain name via DNS queries to the next set of automatically generated domains.

Attackers associate multiple IP addresses with one domain name by rapidly changing the DNS records associated with such name. An address will be registered and then deregistered to be replaced by a new IP address every few minutes or seconds.
Attackers exploit a load balancing technique called **round robin DNS**, and by setting a very short time to live (`TTL`) for each IP address.

Often, some IP addresses will be web hosts that the attackers have compromised.

The host acts as proxies for the attacker's origin server. Round robin DNS associates multiple web servers, each with its IP address, and a domain. When the *authoritative nameserver* receives a query, it hands out a different IP address each time so that no server gets overwhelmed with traffic.
Attackers use this feature to obfuscate their activity.and also set a very short TTL for these addresses as short as 60 seconds. Once the TTL expires, the address will no longer be associated with that domain name.
With **double fast fluxing**, the IP address of the authoritative nameserver is also changed out rapidly.

Optimizing address generation

Given that **density** is the probability that a random address belonging to the set corresponds to a real node, we can define two disjoint subsets:

1. **Local** (*high density*), i.e. similar to the one of the infected node $\longrightarrow$ subnet of the infected node.
2. **Global** (*low density*)

If the ratio between *local* and *global* addresses is:

- ◇ *too low*, the worm may be detected and removed before spreading
- ◇ *too large*, then after infecting all nodes resources are wasted because one node may be infected several times

Note that even low changes in the ratio may be very critical, they create *nonlinear effects*

18.2 A theoretical spreading model

Let's discuss a theoretical model to study the spreading of a worm. The model is epidemiological, meaning that it has been defined to evaluate the overall number of people infected as a function of time:

1. because of a contagious illness
2. in a closed population
3. fully connected population (no distancing)

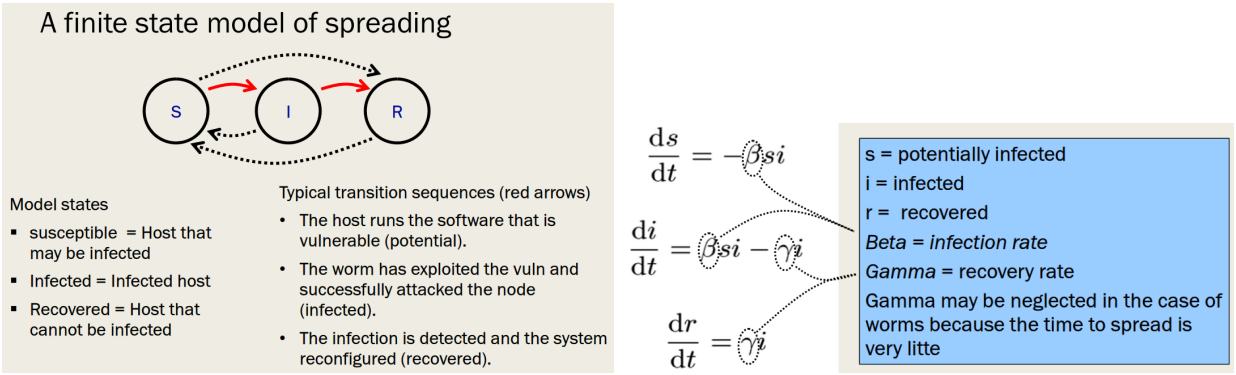


Figure 18.2: Spreading model differential equations

***Kermack** and **McKendrick** model*

- ◇ β is a function of
  - The function to generate the IP addresses
  - The number of the system affected by the vulns
  - It increase with the virulence
  - The model assume that a node can infected any other node = complete connection and no defence
- ◇ γ should not be neglected anytime
  - The spreading is rather slow
  - Patching may be automated (vaccination)
  - There are some automatic components to detect and remove infected nodes

18.2.1 More complex models

Many models have been proposed, to take into discussion also other key points, such as **patching**, and possibly consider how patching impacts on the spreading of a worm.

Another aspect is to define partial connections between nodes, meaning that there is a **topology** which determines how the nodes interact, thus how the spreading may proceed. Consider the internet *Scale Free* topology: When a new node is connected the connection to a , node with a larger number of connections is preferred:

"rich become richer"

- ◇ a few *hubs* with a huge number of connections
- ◇ a huge number of *nodes* with a few connections

This structure is **robust** against random node attacks and faults, since most nodes only have a few connections (hence low spreading capabilities), but is very **fragile** against targeted attacks towards highly connected hubs.

Chapter 19

Challenge II

19.1 Outline

1. Each subject manages a list with its own capabilities
2. The operation field of a capability is encrypted with a key private to the security kernel SK
3. To request operation Op on object O, a subject S sends to SK a message with S, O, Op and the encrypted capability
4. SK decrypts the capability and, if it enables Op on O, it asks O to create a channel with S to execute OP
5. O destroys the channel when Op ends

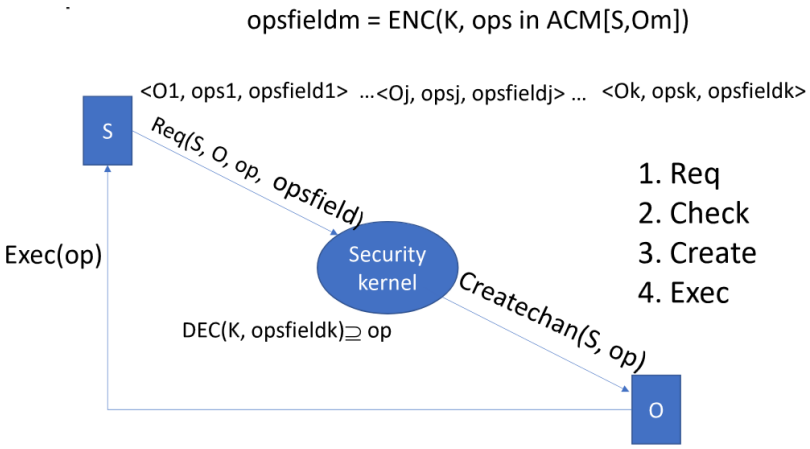


Figure 19.1: Protocol schema

19.1.1 Request

Discover vulnerabilities in the proposed protocol or in the overall system under the assumption that there are no vulnerabilities in the encryption algorithm, i.e. K cannnot be discovered because of mathematical vulnerabilities.

19.2 Proposed Solutions

Below are listed possible vulnerabilities with some form of categorization, although they may be related and/or not mutually exclusive.

Despite the challenge title being *"Capture-the-Flag"*, the following vulnerabilities are –briefly– discussed and addressed from a "security" point of view, however they may be exploited as foundation to define and implement CTF competitions.

19.2.1 Vulnerability 1 - Authentication

First of all we can observe that the lack of **authentication** allows an attacker to impersonate the `Security Kernel`, and thus to invoke `Createchan(S,op)` on objects, choosing arbitrarily `S` and `op` and the destination object.

19.2.2 Vulnerability 2 - Integrity

The attacker, by posing theirself inbetween entities —implementing a *man-in-the-middle* attack—, may manipulate any field on any sent message, with the exception of `opsfield`, since he cannot discover the secret `K`.

In this way he may achieve *Denial-of-Service* or may obtain capabilities he does not own.

"Each subject manages a list with its own capabilities".

Capabilities are usually stored in the subject, but if it's not the case (e.g. distributed systems), depending on where and how such capabilities are stored and managed, an attacker may manipulate them.

19.2.3 Vulnerability 3 - Replay/Redirecting

The attacker may sniff and record valid `Createchan(S,op)` and `Exec(op)` messages and replay them when desired. Even if more costful and complex he may *redirect*[1] them, both obtaining access rights and denying the service for legitimate clients.

He may even copy `op` and `opsfield` and insert them in a `Req` sent by him.

19.2.4 Bonus Vulnerabilities

The following paragraphs do not refer to proper vulnerabilities, but more generally to some security aspects — or to vulnerabilities which are not as crucial/straightforward as the first ones—, which may result in undesired behaviour, possible intrusions, or simply bad system design choices.

Vulnerability 4 - Overflow

Probably not of interest here, but every entity may be flooded by requests and messages, resulting in a denial of service.

Vulternability 5 - Lack of logging

The protocol doesn't explicitly log anything, which instead might be useful to track both legitimate and unlegitimate operation, aiming to discover possible intrusion or to prove in a court of law that an intrusion has taken place.

Vulnerability 6 - Encryption Info Leakage

The problem outline states that `K` cannot be discovered due to mathematical vulnerabilities, so we can leave aside all encryption attacks, mathematical vulnerabilities, etc.

However there are some techniques to infer information starting from the observation of one or multiple encrypted payloads, exploiting statistical properties like size, entropy, and possibly others; the attacker may also infer how costful (hence get an idea of which encryption method is being used) is to encrypt/decrypt fields by observing how much latency is introduced by communications which involve an encrypted payload.

Vulnerability 7 - Least privilege violation

The *least privilege principle* states that a subject should own only the rights he needs to execute the operation he is currently performing.
The described protocol doesn't forbid a subject from requesting multiple operations on the same object before the previous ones have terminated, resulting in multiple channels simulateneously open between an object `O` and a subject `S`, and in multiple rights owned by `S`.
This may be the desired outcome in some situations, otherwise it results in the violation of the *least privilege principle* and might be harmful.

19.3 Baiardi's takeaway message

The *lack* of **identity** and **authentication** is the key issue.

It leads, amongst other problems, to the impossibility to safely **delegate** operations, which is something that capabilities are very useful for; this impossibility arises because if S sends a capability to S' without encryption and

[1] equivalent to recording a message, dropping it, and replaying it towards a different entity

authentication, any attacker can sniff the message and claim that they received the capability and thus are athourized to do perform the requested operation.

Chapter 20

System Design View

20.1 Thesis = Fuzzing Hardware

Baiardi is interested in *Fuzzing Hardware*, he would like to do a thesis on such topic.

20.2 System View

System view is a perspective aiming to consider *design rules* and consequentently *vulnerabilities*, focusing on wrong design choices rather than implementation error. A set of design rules is used to determine the *optimal* set of **controls** for a system, where *optimal* indicates the smallest (i.e. *cheapest* ☺)set of controls to achieve the required robustness. It is important to understand whether the optimal set of controls as imposed by the design rules is compatible with the required performance, and from this point of view we can define a **vulnerability** as a *violation* of system design rules.

20.2.1 Robustness agains Vulnerabilities

Thus, in this scenario, designing a system means finding an acceptable **tradeoff** between *design rules* and *vulnerabilities*.

All the modules in an ideal system satisfy the design rules,and such ideal system is the asymptote of a sequence of distinct systems each applying more controls than the previous one as required by the design rules.
Any difference between the ideal system and the one being created/under analysis may be considered as a **vulnerability**, but to decide if it is an actual vulnerability we consider the **context** and the **cost** of the control against its usefulness.

Some differences between the ideal system and the current one cannot be avoided, supponsing some rules have been violated due to —possibly basic— performance requirements. Other violations (i.e. *missing controls*) instead may be unrelated to performance, hence they should be fixed.
The key strategy to discover vulnerabilities evaluates the cost of missing controls and compares it against

- ◇ the required final *performance*
- ◇ the *risk* ($\mathcal{P}(intrusion) * impact$) due to the missing control

In a whole-system comprehensive view it is important to check **compensative controls**, i.e. a missing control in a module may be compensated by a control in another one.

20.3 Saltzer & Schroeder

S&S Design Principles

1. **Economy of Mechanism**
   The protection mechanism should have a *simple* and small design.
2. **Fail-safe Defaults**
   The protection mechanism should *deny* access by *default*, and grant access only when explicit permission exists.
3. **Complete Mediation**
   The protection mechanism should check *every access* to every object.
   Rather expensive this is the reason is one of the most hard to satisfy.
4. **Open Design**
   Protection mechanism should not depend on attackers being ignorant of its *design* to successfully secure the system. But may be based on the attacker's ignorance of specific information such as passwords or cipher keys.
   "An attacker who learns the key learns nothing that helps them break any message encrypted with a different key. That's the essence of Kerkhoff's principle: that systems should be designed that way."
   However, you should publish information on your system design *iff* it results in a useful **peer review**.
5. **Separation of Privilege**
   The protection mechanism should grant access based on *more than one piece* of information, e.g. *two keys* for a safe.
6. **Least Privilege**
   The protection mechanism should force every process to operate with the *minimum privileges* needed to perform its task.
7. **Least Common Mechanism**
   The protection mechanism should be shared as little as possible among users.
8. **Psychological Acceptability**
   The protection mechanism should be easy to use (at least as easy as not using it).
   Before introducing the last two principles, Saltzer and Schroeder state two things:
   - ◇ Analysis of traditional physical security systems has suggested two further design principles which, unfortunately, apply only *imperfectly* to computer systems
   - ◇ The principles apply both to a system and to the *mechanisms* we introduce to secure the system
9. **Principle of Work Factor**
   Compare the cost of circumventing the mechanism against the resources of a potential attacker.
10. **Compromise Recording**
    Mechanisms that reliably record a compromise of information may replace more elaborate ones that completely prevent loss.
    In other words, if you cannot be robust be aware you may be attacked and be resilient.

20.3.1 Economy of mechanisms

Keep the design as simple and small as possible

KISS rule $\longrightarrow$ *Keep It Simple, Stupid*

Simple implies that less things can go wrong and when bugs occur, they are easier to find, understand and fix. Vulns are proportional to the complexity of a mechanism and the code implementing it. When needed, complexity can be achieved by **composition** of simpler modules, and might be preferreable to building a single more comprehensive yet complex module.

20.3.2 Designing Operating Systems

OS Hardening is an approach embodying this principle: it consists in removing useless OS functionalities for applications of interest.
With respect to kernel, there are two approaches which aim to reduce complexity:

- ◇ **Microkernel**
  Avoid the implementation of complex functions in the kernel, stick only to **basic** ones such as inter-process communication and basic memory management, possibly making room for a *modular* design.
  Unix and Windows do not follow this approach, which is instead adopted by QNX and MINIX, for instance
- ◇ **Esokernel**
  Focus on providing **minimal abstractions** to applications and exposing as much hardware functionality as possible to applications, creating astrong **integration** between the OS kernel and the applications not only violates modularity principles but helps the spreading of errors.

20.3.3 Wrapping Up

- ◇ **Simplify** the interface
- ◇ Complex operations should be implemented by **composing** simple operations

- ◇ If most of the operations are rather complex (and hence powerful), we may be forced to enable a user to invoke a powerful operation just because the simple one it needs is missing
- ◇ Hence, users will apply complex operations even to implement simple operations and this increases their rights (related to the least privilege principle)

20.3.4 Fail-safe and Intel `CSME`

Base access decisions on permission rather than exclusion

Fail-safe related vulnerabilities are fascinating, and may be *non-trivial* to abuse. Some aspects cannot be fixed without replacing the *silicon*, but only *mitigated*.

Fail-safe is related to a *design flaw* baked into millions of Intel chipsets: the problem revolves around cryptographic keys that, if obtained by an attacker, can be used to break the **root of trust** in a system.

Intel Countermeasure

Buried deep inside modern Intel chipsets there is the **Converged Security and Manageability Engine** (`CSME`), which is a miniature computer within a computer with its own CPU, RAM, code in a boot ROM, and with the capability to access the rest of the machine.

Currently, the `CSME`'s CPU core is 486-based, and its software is derived from the free **microkernel** operating system `MINIX`.

You can find a deep dive into the technology behind it all, sometime known as the Minute IA System Agent.

The `CSME` is digital janitor, working behind the scenes, below operating system, hypervisor, and firmware, performing lots of crucial low-level tasks.

- ◇ Bringing up the computer, controlling power levels,
- ◇ Starting the main processor chips
- ◇ Verifying and booting motherboard firmware, and providing cryptographic functions.
- ◇ The engine is the first thing to run when a machine is switched on.

Possible Flaw One of the first things the `CSME` protects is its **own built-in RAM** so that other hardware and software module cannot interfere. However, these protections are disabled by default, there is a tiny timing gap between a system turning on and the `CSME` executing the code in its boot ROM that installs such protections in the form of input- output memory-management unit page tables. During that timing gap, other hardware - physically attached or present on the motherboard can fire off a DMA transfer into the `CSME`'s private RAM to overwrite variables and pointers and hijack its execution. At that point, the `CSME` can be controlled for malicious purposes, all out of view of the software running above it.

Clearly this a strong **race condition**, but still possible.

20.3.5 Complete Mediation

Every access to every object must be checked for authority

This is the most crucial principle when considering performance, since verbatim applying it would be very impactful on the system's performance.

Usually access to object is done only once on the first time, but this leads to possible unauthorized accesses if permissions change later on.
Performance enhancements are achieved by caching results of authority checks, but they should be examined skeptically, to avoid the above mentioned unpleasant situation. Every uncontrolled operation that is a potential vulnerability as it may be invoked without proper rights.

Fail-safe integration

If **both** principles are **satisfied** the system starts in a secure state and, provided that the security kernel is correct, further on only secure transictions are enabled. In other words, it is possible to prove security correctness using an induction approach to determine reachable states.
Fail-safe plays a key role since it makes valid the base case for the induction, it acts as a "starting point". If fail-safe default does not hold then no induction is possible there is no base case for the induction.

20.3.6 Open Design

The design should not be secret
or
Security should not depend on the secrecy of the design or of the implementation

Popularly misunderstood as *"source code should be public"* or *"open-source is safer than proprietary"*.

A system **peer review** is fundamental to discover vulns in design or implementation. A *peer review* is review made by **competent experts** which can discover flaws and vulnerabilities in a system. So, an open source implementation is useful only if it results in a peer review, and if any peer that discovers a vulnerability communicates it to the owner. Instead it is useless and possibly dangerous if there are no peers available to review the code, or they do not reveal to the owner the discovered vulnerabilities.

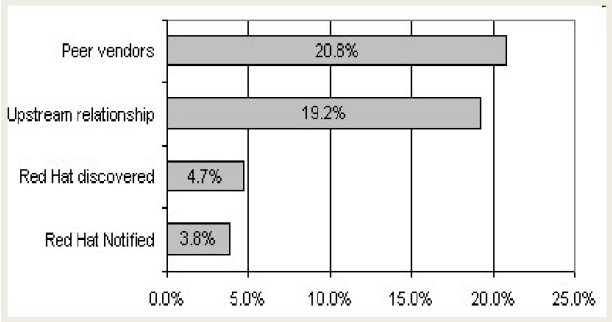


Figure 20.1: RedHat vulnerabilities

As an example, consider RedHat's statistics: as you can see in Fig 20.1, most of the known vulnerabilities were discovered by peers, not by RedHat's own experts.

20.3.7 Separation of Privilege

*Where feasible, a protection mechanism that requires two keys to unlock is more **robust** and **flexible** than one that allows access using a single key*
or
Require multiple conditions to grant privilege

Decomposing

A complex operation should be decomposed into simpler operations, each enabled by a proper access rights, both the complex and the simple ones.
In this way we can check that the subject owns both

1. The right of invoking the complex op (if it still exists)
2. The right of invoking each simple op

20.3.8 Least Privilege

Every subject should operate using the least set of privileges necessary to complete its job or A subject should be given only those privileges it needs to complete its task and only for the time to complete it

According to Baiardi, this is the most crucial principle to be kept in mind. Many intrusion have succeed due to the violation of this principle.

Key Points

- ◇ Owning a useless access right is a *vulnerability*
- ◇ Rights *granted as needed*, *revoked* after being used
- ◇ The `AC` matrix is a *highly dynamic* data structure
- ◇ Rights are *assigned* and *revoked* as the *computation evolves*

This principle should be applied even if the security policy is **static**, since it defines mostly how rights should be managed *after* being granted to each subject rather than how they are granted.
If, in a given time interval, a subject *does not need* a right then this right should be revoked and then granted again later on by updating the `ACM` to prevent the subject from exploiting the right in such time interval.
The right is removed at the beginning and then granted at the end of the *"not-needing" interval*.

Protection Domain Switching

Protection Domain Switching (`PDS`) consists in updating an `ACM` row, making the same subject executed with *updated rights* in `ACM`.
Note that it is possible to have a `PDS` without a *context switch*. The performance overhead introduced by `PDS` is a function of the implementation level and the adopted implementation of the `ACM` (*Capabilities* vs `ACL`); When the implementation relies on capabilities revoking a right is not trivial, delegation and race conditions are involved.

Small Domain Principle

An alternative definition of this principle is **the small domains principle**, to stress the importance of minimizing the protection domains $\longrightarrow$ the number of the subject rights $\longrightarrow$ the *frequency* of the *commutation* of the protection domain.
Besides, as the size of the protection domain decreases, it also decreases the risk of an attack against the considered subject.
If the protection domains are *not small*, then we revoke access rights when not needed and grant them again when needed, as described before.

Notice the strong relation between the two last principles because segmented networks force the reduction of protection domain and separation of duties enables the implementation of the least privilege.

Common implementation

In the classical solution a *domain switching* occurs when

1. A procedure (method) is **invoked**
2. A procedure (method) **returns**

Instead of updating row, creation (*invoke*) and deletion (*return*) are typically preferred.
When a procedure is invoked, a new row that defines its rights is created and gets deleted when the procedure returns.
Rights are paired with the **instance** of a procedure executed by (or on behalf of) a subject rather than with the actual procedure code or with the subject.

The rights in the new row are a function of:

- ◇ Method private variables
- ◇ Input parameters

The program structure in terms of *classes/methods* defines the strategy to manage rights granted to the subjects on program data structures, allowing automated handling of `PDS` and the programmer to choose the size of each protection domain.

Network Programming

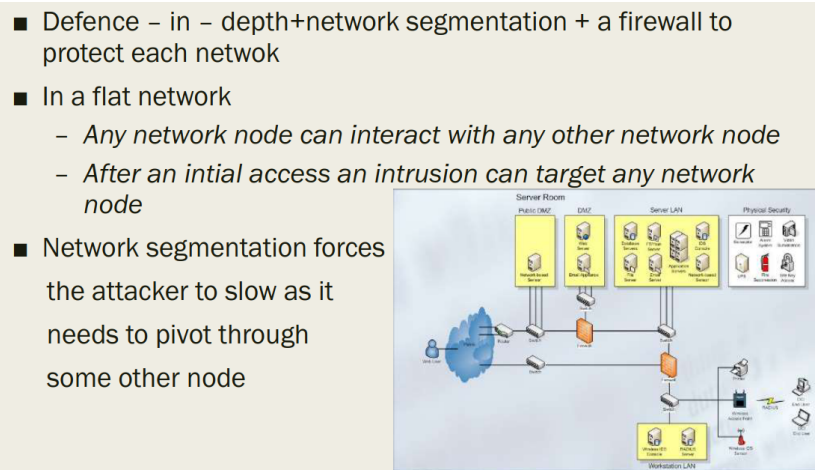
In network programming the least privilege principle is managed through dynamic communication channels.
A server can be designed by introducing a distinct communication channel for each operation implemented by the server; a process is **connected**[1] to the channel paired with an operation `Op` —with a service `S`— when it can invoke the operation —can access the service—; then process is **disconnected** from the channel when it can no longer invoke the operation.

The adoption of dynamic channels *reduces* the **overhead** to reject a request by implementing in the OS kernel a mechanism to ope and close ports.

[1] i.e. can send messages

The overhead *on the server* can be further reduced by *"spreding the news"* that a message from a given node to a given port of a receiver can be *dropped* by routers or firewall; However, this implies a larger overhead on other nodes to update *routing tables* or *firewall rules*.
On the other side, if only the final node or, even worse, the final server can drop messages we offer an opportunity for a **DDoS attacks** where a server is overflown by a large number of requests to be discarded.

Defence-in-depth and Network Segmentation



Network Segmentation can be seen as an implementation of *Defence-in-Depth* (Least Privilege key point) when designing system networks.

Figure 20.2: Least Privilege and Network Segmentation

ZeroTrust Network

Recall that the basic idea of `ZT` is to authenticate both the **user** and the **device** exploited by the user to access a **service**;

Authenticating a device means searching for the device security status in an inventory to check the vulnerabilities that affect the device and which patches have been applied

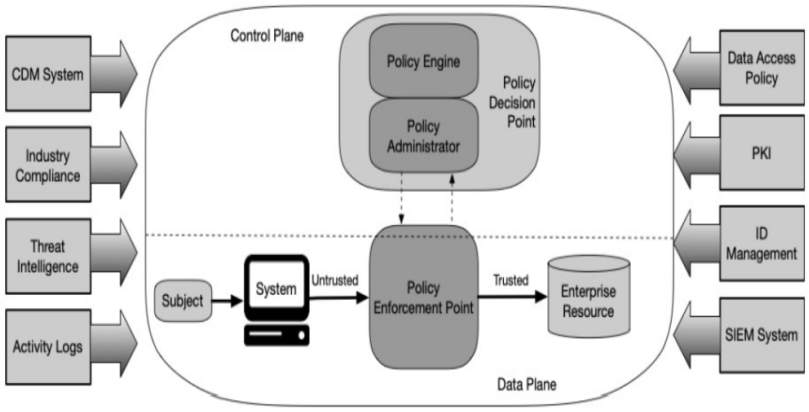


Figure 20.3: ZeroTrust in Networks to apply *Least Privilege* principle

1. **No implicit trust** is granted to assets or user accounts based *solely* on their physical or network location (i.e., local area networks versus the internet) or based on asset ownership (enterprise or personally owned).
2. Authentication and authorization (both subject and device) are performed **before** establishing a session to enterprise resources
3. Zero trust is a response to enterprise network trends that include remote users, "bring your own device" and cloud-based assets not located within an enterprise-owned network boundary.
4. Zero trust protects **resources** (assets, services, accounts, etc.), ***not* network segments**, since the network location in most nowdays infrastructures is no longer the prime component to the security posture of the resource.
5. All data sources and computing services are considered resources
6. All communication is secured regardless of network location because location alone does not imply trust.
7. Access to individual enterprise resources is granted on a per-session basis (no caching). Trust in the requester is evaluated before granting access with the least privileges to complete the task.

8. Access to resources is determined by dynamic policy (= the observable client identity, application/ service, and the requesting asset) and may include behavioral and environmental attributes.
9. The enterprise monitors and measures the integrity and security posture of all owned and associated assets. No asset is inherently trusted.

*"You cannot satisfy **ZeroTrust** unless you satisfy, at least to some degree, the **Least Privilege** principle."*

20.3.9 Least Common Mechanism

Minimize the amount of mechanisms common to more than one user and depended on by all users

Instead of **sharing** *mechanisms* theirselves, the implementation usually relies on making the information flow along *shared channels*, allowing the creation of **covert channels**.
Sharing should be avoided since it also decreases **isolation**, and is difficult to manage when handling Virtual machines and Host/Network Segmentation.

Usually shared mechanisms are rather **powerful**, hence they should be **decomposed** into simpler ones to better satisfy the separation of privilege and least privilege principles, since simpler operations allow to assign to each subject *only* the rights it *needs*, and it is entitled to.

Cover Channels

Covert Channels are a means of communication between two processes that do not explicitly provide interaction mechanisms. One process is a *Trojan*, which transmits data covertly and the other a *Spy* which receives data.

1. **Storage Channels**
   Communication is achieved by modifying a stored object.
   Countermeasure $\longrightarrow$ shared memory areas should be always *reinitialized* before being passed to another process, and they should not exist among processes with distinct security levels.
   With cache memories information can be transmitted through faults (a page has been accessed or not)
2. Information is transmitted by affecting the relative timing of events.
   The `Hi` *Trojan* process transmits "we attack at dawn by attempting to acquire the disk drive at midnight";
   the `Lo` *Spy* process knows that, if the disk is unavailable at midnight, then "we attack at dawn", otherwise, not.
   Timing channels are very difficult to avoid on time-shared system but are **noisy**. *Countermeasure* $\longrightarrow$ Traffic Analysis and Pattern Recognition, to detect *noise*.

20.3.10 Psychological Acceptability

The human interface should be designed for ease of use so that users routinely and automatically accept the protection mechanisms correctly
or
Do not adopt policies users will surely violate

Security mechanisms should allow ease of installation, configuration and use, and in general should not increase the complexity of accessing a resource: such complexity should be hidden.

20.3.11 Salter & Schroeder considerations

After defining the first 8 principles, *S&S* admit that

Analysis of traditional physical security systems have suggested two further design principles which, unfortunately, apply only **imperfectly** to computer systems

1. Principle of **Work Factor**
   Compare the cost of circumventing the mechanism with the resources of a potential attacker
2. Principle of **Compromise Recording**
   Mechanisms that reliably record a compromise of information may replace more elaborate ones that completely prevent loss
   i. Robustness vs resilience
   ii. If you cannot be robust be at least resilient
   iii. At least discover successful intrusions and persistence

The two principles are useful if some attacks are successful in spite of the adoption of the previous principles, and

can be even more useful if some of such principles have been violated. In some sense they anticipate the presence of vulnerabilities and possible failures, reminding a system designer to not believe that they can be robust against *any* adversary.

20.3.12 Work Factor

Compare the cost of circumventing the mechanism with the resources of a potential attacker: the enemy is the master and the measure

The probability of a successful attack *increases* with the **resources** the attacker can access, while the cost of circumventing a mechanism is the **work factor** of the attacker.
So, we can consider a mechanism better than another if it can be defeated only through a larger amount of work.

The number of attacks in escalations and their attribute together with the action to collect the information on the vulnerabilities and implement the attacks determine the *(minimal)* attacker work.

Measuring the amount of work the attacker (the adversary) has to do is an otput of **adversary emulation**, i.e. mimicking the attacker to understand if they can successfully attack a system. If you have information on your adversaries then you can exploit *Att&ck Matrix* to understand if and how they can successfully attack your system, otherwise

20.3.13 Compromise Recording

Mechanisms that reliably record a compromise of information may replace more elaborate ones that completely prevent loss

A mechanism supports the discover of unauthorized use if it produces a **tamperproof record** that is reported to the owner. It is difficult to guarantee discovery after a computer system has been attacked and controlled from the outside. Logical damage (and internally stored records of tampering) can be undone by a clever attacker.

When collecting for security the integrity of the log is fundamental, if the integrity of a log for security is not assured then the log is useless.
The difference between a sequence of blocks and a blockchain is a good example of the difference between collect for debugging and collect for security

20.3.14 Takeaway message

Remember that tools do not solve security problems. Buying a firewall doens't make a network secure, and it's not the key point. The real problem is to understand and define the policy rules to be applied.

Chapter 21

Google System Design Strategies

21.1 Least Privilege

Users should have the minimum amount of access to accomplish a task, regardless of whether the access is performed by humans or systems.
Besides, since accounts may be compromised, systems should limit user access to the data and services required to do the **task at hand**.

These restrictions are most effective when you add them at the beginning of the development lifecycle, during the design phase of new features

security by design, as required by GDPR

Unnecessary privileges lead to a growing surface area for possible mistakes, bugs, or compromises, i.e. creating security and reliability risks that are expensive to contain or minimize in a running system.

In other terms:

- ◇ it is better to reduce the attack surface in the **design** because this reduces the cost more than an update after deploying a system.
- ◇ the cost of security mechanisms depends upon the time when they are adopted, i.e. *the later the more expensive*

You should

- ◇ limit the cost of controls by prioritizing what to protect. Not all data or actions are created equal, and the makeup of your access may differ dramatically depending on the nature of your system.
- ◇ not protect all access to the same degree
- ◇ apply the most appropriate controls and avoid an *"all-or-nothing"* mentality

You need

- ◇ to classify access based on impact, security risk, and/or criticality.
- ◇ to handle access to different types of data (public versus company versus user versus cryptographic secrets) differently
- ◇ to treat administrative APIs that can delete data differently than service-specific read APIs.

Chapter 22

Data Handling

22.1 Classification

Do understand how to protect data several security policies **classify data** into *four* categories according the need for **confidentiality** and the resulting risk profile:

Security needs classification

1. **Public Data**
   Data can be disclosed without restriction.
   Directories, Maps, Syllabi and Course Materials, de-identified data sets
2. **Internal Data**
   Confidentiality of data is preferred, but information contained in data may be subject to open records disclosure.
   email correspondence, budget plans, employee EmplID
3. **Sensitive Data**
   Data confidentiality required by law, policy, or contractual obligation
4. **Restricted Data**
   Restricted data requires privacy and security protections. Special authorization may be required for use and collection.
   data sets with individual Social Security Numbers (or last four of SSN), credit card transaction or cardholder data, patient health data, financial data

Another classification is the one based on the availability needed for data:

Availability needs classification

1. **Supportive Data**
   Data that is necessary for day-to-day operations, but is not critical to the mission or core functions.
2. **High-priority Data**
   Availability of data is necessary for organization function. Destruction or temporary loss of data may have an adverse affect on organization mission, but would not affect organization-wide function.
3. **Critical Data**
   Critical data has the highest need for availability. If the information is not available due to system downtime, modification, destruction, etc., the organization functions and mission would be impacted. Availability of this information must be rigorously protected.

22.2 Protection

Having classified the data, we can discuss how to protect it:

1. Data **discovery and inventory**
   The first step in data protection is discovering data sets in the organization, paying attention to which are business-critical and which contain sensitive data that might be subject to compliance regulations.
2. Data **loss prevention** (**DLP**)
   DLP is a set of strategies and tools to prevent data from being stolen (exfiltration), lost, or accidentally deleted. DLP besides protecting against data loss, also helps to recover it.
3. Storage with **built-in data protection**
   Modern storage equipment provide *built-in* disk clustering and redundancy `RAID`. For example, some store

providers offer up to 14 nines of durability, low cost enabling storage of large volumes of data, and fast access for minimal $RecTimeO/RecPercObj$.

4. **Backup**
   A backup is copy of data and stored separately from the source; allows to restore the data to a previous state in case of loss or modification.
5. **Snapshots**
   A snapshot is a backup of a complete image of a system, including data and system files. A snapshot can be used to restore an entire system to a specific point in time.

22.2.1 RAID

Redundacy through Array of Inexpensive Disks

- ◇ **RAID 0: Striping**
  Data is striped across multiple disks to improve performance, but there is no redundancy. If one disk fails, all data is lost.
- ◇ **RAID 1-4: Mirroring**
  Data is mirrored on two or more disks for redundancy. If one disk fails, the data is still available on the mirrored disk(s).
- ◇ **RAID 5: Striping with Parity**
  Data is striped across multiple disks with distributed parity for fault tolerance. If one disk fails, data can be reconstructed from parity information on the remaining disks.
- ◇ **RAID 6: Striping with Double Parity**
  Similar to RAID 5, but with an additional set of parity data, providing fault tolerance against the failure of two disks simultaneously.
- ◇ **RAID 10: Mirrored Striping**
  It combines mirroring and striping. Data is both mirrored and striped for performance and redundancy. It requires a minimum of four disks.
- ◇ **RAID 50: Striped RAID 5 Arrays**
  It combines the straight block-level striping of RAID 0 with the distributed parity of RAID 5. It requires a minimum of six disks.
- ◇ **RAID 60: Striped RAID 6 Arrays**
  Similar to RAID 50, but with double distributed parity for enhanced fault tolerance. It requires a minimum of eight disks.

1. **Replication**
   Copying data *periodically* on an ongoing basis locally or to another location, providing a living up-to-date data copy, allowing both recovery and failover to the copy if the primary system goes down.
2. **Encryption**
   Altering data content according to an algorithm that can only be reversed with the right encryption key. Encryption makes data unreadable, protecting it from unauthorized access even if stolen.
3. **Data erasure**
   Eraruse limits liability by deleting data that is no longer needed. This can be done after data is processed and analyzed or periodically when data is no longer relevant. Erasing unnecessary data is a *requirement* of many compliance regulations, [a].
   Effectively erasing data is not as trivial as it may seem
4. **Disaster recovery**
   Disaster recovery is A set of practices and technologies that determine how to deal with a disaster, such as a cyber attacks, natural disasters, or large-scale equipment failures. *Disaster recovery* typically involves setting up a remote disaster recovery site with copies of protected systems, and switching operations to those systems in case of disaster.

***Protection** Solutions*

[a] such as GDPR

22.3 API

The **API** of a distributed system includes all the ways someone can *query* or *modify* its **internal state**.

Administrative `APIs` are clearly more critical than "user" ones for what concerns reliability and security: typos and trivial mistakes by an admin may result in catastrophic outages or leaks of huge amounts of data, making such APIs very *appealing for attackers*.
Administrative APIs include APIs for *installing/removing* software and deploying containers or VMs, and some **maintenance** and emergency APIs to delete corrupted user data or state, or to restart misbehaving processes.

When designing with the *Least privilege principle* in mind, the most intuitive solution is to **decompose** a large API in smaller ones. The **POSIX** API, popular for its flexibility, is huge and kind of suffers from this point of view; for instance, traditional host setup and maintenance is often performed via an interactive `OpenSSH` session or with tools that script against the `POSIX API`, in both cases exposing the whole API to the caller, making it difficult to constraint user activities only to the ones he actually needs.

However, in some situations, minimal APIs may prevent an admin(/user) from performing some tasks. ***Breaking-glass** Mechanisms* (BGM) have been created to this extent, essentially allowing an admin to bypass –violate– access control mechanisms [1] and execute critical operations such as shutting down a machine or killing a process.
BGM may consideratively speed up problem solving and error recovering, but their usage should not be *abused*; actually it should be **ruled** by the security policy, and recorded –*logged*– and checked –*audit*–.

The ability to use a BGM should be *highly restricted*. In general, it should be available only to the team responsible for the operational **SLA**[2] of a system.
In *ZeroTrust* networks, BGMs should be available only from specific locations, e.g. panic rooms, resulting in a strategy which **distrusts** *network location* but **trusts** a few locations with additional **physical access controls**.

BGMs kind of subvert the *ZeroTrust* approach, so they should be used only as a debugging mechanisms or in extreme cases

BGMs should not only be logged and check (*audited*), but also regularly tested to ensure that they work when actually needed. Besides, after solving an issue via a BGM, the security team should **inspect** the underlying problem and provide a solution to avoid needing to use a BGM in case such problem arises again.

Service Level Agreement Each service contract determines performance indexes to evaluate quality of service, and the minimum performance the service **must** provide. Some indicators for a **SLA** focused on performance are:

- ◇ *Network performance* (bandwidth, **availability**)
- ◇ System *component availability* (mainframe, network, bandwidth, ...)
- ◇ End-to-end *service availability*
- ◇ Web *response time*
- ◇ **Uptime**

22.4 Yale - BGM Guidelines

1. *Break-glass* solutions are based on **pre-staged** emergency user accounts (created in advance, to carefully define respective access controls), managed and distributed in a way that can make them quickly available without unreasonable administrative delay. In other words they should be *simple*, *effective*, and *reliable*.
   Account *usernames* should be simple and **meaningful**, while *passwords* should be **effective** but **easy to remember** by the admins.
2. *Account Permissions* should be set to ***minimum** necessary privilege*.
3. **Auditing** should be enabled if available, to log details of the account usage and details of the work carried out while using the account.
4. The individuals who create the accounts are **not** those ones reviewing the audit trails, since this can be a source of abuse.
5. The *break-glass* accounts and distribution procedures should be **documented** and tested as part of implementation.
6. BGMs require the emergency-account details to be made available in an appropriate and reasonable manner: such details may be provided on a media e.g. a printed page, a magnetic-stripe card, a smart card or a token. Some distribution possibilities for break-glass accounts include the following:
   i. Kept behind **actual glass** in a cabinet, where access to the accounts literally requires to break the glass [3], providing an obvious indication that the accounts have been accessed and a deterrent to casual use.
   ii. Maintained within **sealed envelopes**, where a broken seal would be an obvious indication that the accounts have been accessed.
   iii. **Locked** in a desk drawer that only *specific people* can access.
   iv. Sealed and taped to the side of a monitor in a criical area, **visible** to many so it will be obvious when it is missing or damaged.
   v. For cases where *more than one person* is needed to declare an emergency, **locked** in a safe or cabinet where one person knows the combination or has the **cabinet key** and a different person has the **room key**.

[1] i.e. the Access Control Matrix
[2] *Service Level Agreement*
[3] similar to a fire extinguisher or alarm

22.5 Testing

A system designed for least privilege is more complex than one that grants any privilege to anyone, so **testing** is needed. Sadly, also testing is more complex too, because a problem (i.e. an error) arises any time an *access right* is missing. Testing and debugging to discover **legal rights** that are *not* granted both **differ** from those to check the correct implementation of functions.

A **testing infrastructure** monitoring a deployed system may be adopted but two problems follow:

1. *Testing **of*** least privilege to ensure that access is properly granted only to *proper resources.*
2. *Testing **with*** least privilege to ensure that the *infrastructure* for testing has only the access it needs.

22.6 Authentication & Authorization

Authenticate $\longrightarrow$ verify the *identity* of a user or process as we have seen
Authorize $\longrightarrow$ evaluate (to permit) a request from an *authenticated* party

Authorization may consider additional parameters aside from *identity* and *operation* requested, such as:

1. Requesting **device**
2. Request **parameters**
3. Metadata about **geolocation** of the device, **security status** of the device

For both *MultiParty authorization* (`MPAuto`) and *3-Factor Auth/Auto* (`3F`), request authorization is confirmed by **another node**, beloging to a subset of more robust and secure node with the role of confirming choices of less robust. An interesting case is when the "more robust node" is a *smartphone*, which makes `3FA` very useful when a malware on a **workstation** tries to implement an attack, but useless against **insiders** because they can freely use their smartphones.

Temporary Access consists in granting for *minimal time* rather than with minimal access rights, and is useful when fine-grained controls are not available for every action to grant the least privilege possible with the available tools. Temporary access can be granted in a structured and scheduled way or in an *on-demand* fashion where users explicitly request access, as it happens with "*Run as administrator*" or "`sudo`".

22.7 Summary - Designing for *Least Privilege*

To sum up, let's discuss what we need to design a system according to the least privilege principle:

1. A **comprehensive** knowledge of your system to classify each part according to its ris
   classification is much easier in the design phase
2. Based on this classification, a **partitioning** of your system and access to your data to as fine a level as possible, allowing **small functional APIs**, which are a necessity for least privilege.
3. An **authentication** system for validating credentials as users attempt to access the system.
4. An **authorization** system that enforces a well-defined security policy for your finely partitioned systems.
5. A set of advanced controls for **authorization** to provide *temporary*, *MFA* and *MPA* approval.
6. A set of operational requirements for your system to support these key concepts/abilities:
   i. to **audit** all access and to generate signals so you can identify threats and perform historical forensic analysis
   ii. to design, define, test, and debug your **security policy**, and to provide end-user support for it
   iii. a **Breaking-glass** mechanism to help when your system does not behave as expected

22.8 Design for Robustness

We can integrate the features for **system robustness** (i.e. *S&S*) we have already seen with rules for understandability, for several reasons:

1. *Decreases the likelihood of security vulnerabilities or resilience failures.*
   Any system change might introduce a new vulnerability or compromise the resilience.
   $\longrightarrow$ The less understandable the system, the more likely this occurs (`KISS` rule)
2. *Facilitates effective incident response.*
   In an intrusion, it is vital that responders can quickly and accurately assess the damage, contain the incident, and identify and remediate root causes.

⟶ A complex, difficult-to-understand system significantly hinders this response.

3. *Increases confidence in assertions about system security*
   Assertions about security are system **invariants** *rightarrow* properties that must hold for all possible system behaviors.
   As an example, when receiving malformed or malicious inputs, the system behavior must not violate any security property. In a difficult-to-understand system, invariants may be impossible to verify.
   ⟶ Only an abstract reasoning can establish invariants and *not* testing.

22.9 Invariants

The system has to ensure that a desired invariant holds even if its environment misbehaves in unexpected or malicious ways, taking into account that also the environment might misbehave, since it also includes everything the system cannot directly control, from nefarious users who submit maliciously crafted requests to hardware failures that result in random crashes (*"chaos engineering"*).

Invariant examples

- ◇ Only authenticated and authorized users can access the persistent data store.
- ◇ ll operations on sensitive data in the persistent data store are recorded in an audit log in accordance with the auditing policy
- ◇ The number of queries to a back-end scales with those the front end receives.
- ◇ The system can only receive RPCs from a set of designated systems and can only send RPCs to a set of designated systems.

Natural language can be **formalized** by pairing a user with state variables denoting properties indicating whether the user has been authenticated and authorized.

22.9.1 Analyzing Invariants and Understandability

A **spectrum** defines the *tradeoff* between the impact resulting from the violations of an invariant and the effort to assure it actually holds.
The problem can be approached from two perspectives; The low-effort/high-performance is **testing** to remove bugs that could violate the invariant. The other is analyzing on formally proving soundness, having the system properties **formally modeled**; this approach is in most cases unfeasable and extremely costful even for mid-sized systems.

A different approach is to define **mental models** of a system which abstract and focus on some system features neglecting secondary ones, and then design the system such that it detects and signals anomalous situations outside the mental model.
Mental models are defined starting from the most frequent cases of *functioning* and interaction, while security cannot be based on past experience, since it is linked to intrusions, which are **rare events**.

We can improve **understandability** by **decomposing** a system horizontally and vertically through modules and onions, creating a hierarchy of virtual machines.
Decomposing verticaly simplifies many but not all problems, because security and robustness are holistic: a mistake in one level can enable attacks in higher levels.
Similar problems arise in horizontal decomposition into modules where centralization reduces complexity by using a single authentication server or by a strong synchronization between modules.

22.10 Identities

Identities are important not only for users but even for services (logging, audit). Besides recall that *Access control* robustness is built on top of user identity.

IPs are dynamic and can be reused, making them not reliable identifiers; besides they can be easily spoofed to produce fake messages.

Identities properties

- ◇ **Understandable** by a human
- ◇ Robust against **spoofing** (no vs password vs certificate)
- ◇ Not **reused**
  if a person working in an organization leaves the organization reusing its identifier may result in attribution problems in log analysis

22.10.1 Google

As an example let's see what are identities in Google:

1. Users
2. Administrators
3. Machine
   i. The DNS name is the identifier
   ii. The identity describe the role of the machine (testing, users, admin...)
4. Workload
   i. Required by a user
   ii. Assigned to a machine in a predefined set
   iii. Handled by a kubernetes like system that solves among others
5. Load balancing
6. Resource optimization

22.11 Trusted Computing Base

Recall that the **Trusted Computing Base** is the set of components (hardware, software, human, ...) whose *correct functioning* is sufficient to ensure that the security policy is **enforced**, or, conversely, whose *failure* could cause a **breach** of the security policy.

The TCB must handle also misbehaving by component outside it. The interface between the TCB and "everything else" is referred to as a **security boundary**, allowing the TCB to treat anything that crosses the security boundary with *suspicion*.

Ensuring that a system enforces the desired security policy, implies understanding that the whole TCB is relevant to such security policy. Understanding a TCB becomes more difficult as it includes more code and complexity, so it is important to keep TCBs as small as possible and to exclude components not involved in upholding the security policy. Sometimes this aligns with **database decomposition** where data with *distinct security requirements* is mapped into *disjoint databases*. Furthermore there usually a **frontend** which allows users to access data(bases): errors in the frontend may lead to security violations; Hence, in a web threat model, the TCB again includes all the frontend, and it may be useful to decompose also the frontend into multiple ones which respectively access a disjoint database.

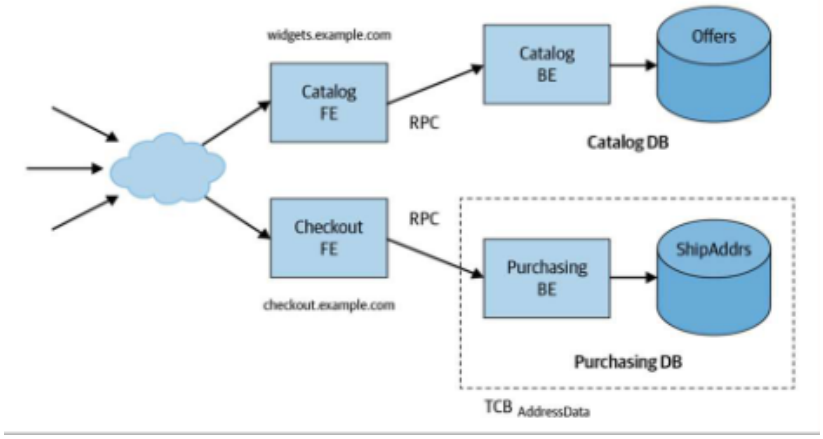


Figure 22.1: TCB Decomposition

22.12 Design for Change

"*Design for Change*" i.e. design to allow changes to disable new *vulnerabilities*. This poses many software engineering issues, but focusing only on security ones, we can first consider which attributes should *changes* —in a system— have:

1. Incremental
2. Documented
3. Tested
4. Isolated
5. Qualified
6. Staged i.e. check the benefit of the n-th change before applying the n+1-th change

Note that Zero-day are not the only critical vulnerabilities; important questions are:

- ◇ "is this vulnerability currently exploited?"
- ◇ "is this vulnerability in the attack surface or can it be reached from the surface = an attack chain from the surface?"

Based on the answers to the previous questions, a **patching policy** chosen; Generally patches are distributed using two distribution channels

1. **low** risk vulnerabilities ⇒ according to scheduling policy
2. **high** risk vulnerability ⇒ immediate patch to minimize the ris

22.13 Design for Resilience

Each system layer should be designed to be **independently resilient**, implementing defense in depth among the layers.

Compute the cost of prioritizing each feature, in order to understand which features are critical enough to be sustained and which are less important and can be throttled or disabled.
Compartmentalize the system along clearly defined boundaries to promote the independence of the isolated functional parts, achieving both *segmentation* and *isolation*.

Reduce *reaction time* by **automating** as many of your resilience measures as possible, exploiting intrusion detection, anomaly detection and pattern matching; Also periodically validate resilience properties to ensure the effectiveness of the system.

22.13.1 Risk assessment perspective

Both attackers and defenders assess a target for weaknesses, but while *attackers* perform **reconnaissance** against targets, find weaknesses, and model their intruisons and the related attacks, the *defenders* instead (should) **know** the system, allowing for the assessment to be much more precise and comprehensive, but the defender has to implement *remediations*, not attacks, which is very different. The crucial point here is to understand that an *attacker* is **not** a *defender*.

Defense is more effective if the defender understands the attacker methods and the more the defender understands the attacker methods[4] the more the benefit is amplified.

"Enemy is the master"

22.13.2 Degradation

The general idea is that if a system cannot resist to adversaries, then it can –and should– **limit** the blast radius of attacks.

The system layers are complementary, with each layer anticipating the weak points or likely failures of the previous on; designing for defense in depth assumes system components or even entire systems can fail due to physical damage, hardware malfunction, or an intrusion.

To control degradation, select properties to disable when specific circumstances arise, while doing all you can to protect the system security. By designing multiple response options for these events, the system can have controlled ***proactive*** **breakpoints** rather than a chaotic collapse; in other words, we can "design" even *crashes*.

For what concerns the "**blast radius**", controlling it means *compartmentalizing* the impact of an event, as, for instance, compartments on a ship grant resilience against the whole ship sinking; security breach or traffic overload in one compartment should not jeopardize all compartments. To this extent, when designing for resilience, there should be **compartmental barriers** that constrain both attackers and accidental failures, creating **failure domains** and allowing to better tailoring and automation of responses.

The principle behind compartmentalization is to isolate system elements and enable and control the interactions essential for their intended purpose. Compartmentalization can happen in two ways:

[4] MITRE ATT&CK matrix

1. Parallel decomposition
2. Clustering
    i. Containers
    ii. VMs
    iii. Phyisical machines

22.14 Trusted Platform Module

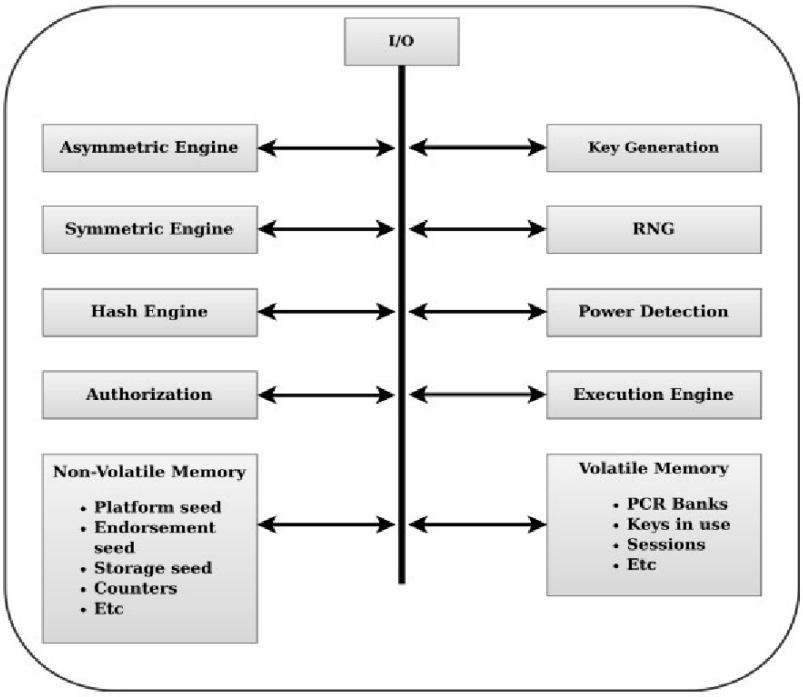


Figure 22.2: TPM

Usually this protected chips are used to perform the **hash** of software running on nodes, to ensure that the software has not been manipulated, allowing a system to **trust** another one. More generally , *Remote Attestation Procedures* (`RAT`s) forsee one an *Attester* peer which produces believable information –*Evidence*– about itself to enable a remote *Challenger*[5] peer to decide whether the *Attester* is a trustworthy peer or not; `RAT`S are facilitated by an additional vital party, the *Verifier*.

22.14.1 AWS Nitro System

An EC2 server is made up of one or more **Nitro Cards** and a main system board that contains the host CPUs and memory.

Nitro Cards are dedicated hardware components with 64-bit processing capabilities and specialized *Application Specific Integrated Circuits* (`ASIC`s) that operate independently from the system; they implement all the outward-facing control interfaces used by the EC2 service to provision and manage computer, memory, and storage.

Three key components of the Nitro System, provide faster innovation, enhanced security, and improved performance:

1. *Purpose-built Nitro Cards*
   Hardware devices designed by AWS that provide overall **system control** and input/output (`I/O`) virtualization independent of the main system board with its CPUs and memory.
2. *The Nitro Security Chip*
   Enables a *secure boot* process based on a hardware **root of trust**, the ability to offer bare metal instances, as well as defense in depth to protect the server from unauthorized modification of system firmware.
3. *The Nitro Hypervisor*
   A minimized and firmware-like **hypervisor** to provide strong isolation, and performance nearly indistinguishable from a metal server. A limited and carefully designed component intentionally minimized and built with the capabilities to perform its required functions, and no more.
   The meticulous exclusion of non-essential features from the Nitro Hypervisor eliminates entire classes of bugs

[5] aka "Relying Party"

that other hypervisors can suffer from (remote networking attacks or driver-based privilege escalations). Besides, even in the unlikely event of a bug in the Nitro Hypervisor that allows access to privileged code, there is an **inhospitable environment** to any potential attacker due to the *lack of* standard operating system features such as interactive shells, filesystems, common user space utilities, or access to resources that could facilitate lateral movement within the larger infrastructure
(harden your system *S&S*)

In the Nitro System there is a **Nitro Controller** which provides the hardware **root of trust** for the overall system. It is responsible for managing all other components of the server system, including the firmware loaded onto the other system components; It is the exclusive gateway between the physical server and the control planes for EC2, EBS and VPC, and it operates together with Nitro Cards as a single domain.

A **Security chip** links the domain of the Controller to the domain of the external component that controls the system's main board (where the security chip usually resides).

By design there is **no operator** access, meaning there is no mechanism for systems or persons to log in to EC2 Nitro hosts, access memory of EC2 instances or any customer data stored on local encrypted instance storage or remote encrypted EBS volumes.
If any AWS operator, even with the highest privileges, needs to do maintenance work on an EC2 server, they can only use a limited set of authenticated, authorized, logged, and audited *administrative APIs*. *None* offers access customer data on the EC2 server.
Clearly this leads to not-so-easy debugging operations.

22.15 Other resilience concepts

22.15.1 Graceful Degradation

Essential and difficult choices should be carefully made in advance, not during an incident while being "under pressure". Free up resources, thus decrease failed operations, by disabling infrequently used features, least critical functions, or high-cost service capabilities, prioritizing instead important features and functions. Improving the system capacity to absorb load or failure supports all other resilience mechanisms—and buys more time for response.

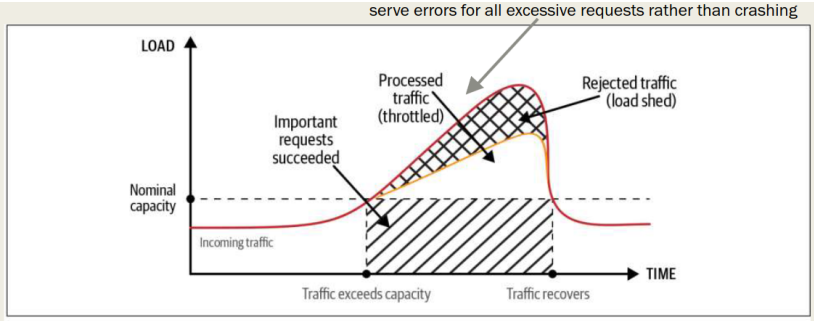


Figure 8-4. Using load shedding and throttling to manage a load spike

22.15.2 Failure Management

Failure management should balance between optimizing for reliability by failing *open* (safe) and optimizing for security by failing *closed* (secure).
To maximize **reliability**, a system should resist failures and serve as much as possible in face of uncertainty, even if integrity is not intact.
$\Longrightarrow$ If ACLs failed to load, the default ACL is *"allow all"*

To maximize **security** instead, a system should lock down *fully* in uncertainty. If it cannot verify integrity the system can't operate and should protect itself as much as possible.
$\Longrightarrow$ If ACLs failed to load, the assumed default ACL is *"deny all"*

"Emergency exit doors" need *fail safe* rather than *fail secure* ones.

Security-critical operations should not *fail open* because an attacker would be able to degrade system security through simple DoS attacks(think of a firewall), but a controlled degradation of security-critical operations is possible. A lower-cost component could replace failing security controls apllying stronger (possibly simpler) controls, making attacks to the default security controls *not* effective, enhancing **resilience**.

A strategy adopted by Google is to limit an attacker's impact using the **location** hosting a service to prevent an adversary who has compromised a datacenter to read data in other ones: *location-based* cryptographic compartments limit access to applications and their stored data to specific locations, containing the *blast radius* of attacks.
Aside from databases, also service might also reject requests from locations it doesn't expect to communicate with.

However note that a known location must *not* imply **trust**, authentication and authorization must be granted anyway.

22.15.3 Failure Domains

To address component failures, system design must incorporate **redundancies** and distinct **failure domains**, to prevent the failure in one domain from affecting components in other domains.

Failure domains achieve functional isolation by partitioning a system into multiple, equivalent but completely independent copies.
Operating failure domains and maintaining their isolation requires ongoing effort but these domains increase system **resilience** in ways other *blast radius* controls *can't*.

Some **bad data** may be present at the data source or within individual failure domains, forcing each failure domain's own data to be functionally independent from other domains.
To address these problem there are two main approaches, one is to restrict how data updates enter a domain through **validation checks**, while the other is to enable systems to write the last known good data to disk to increase –long-term–resilience to losing access data, implementing *defense in depth*.

22.15.4 Takeaway and Observations

Let's sum up and consider the takeaway messages:

1. Even just two failure domains provide A/B testing capabilities and limit the blast radius of changes to one domain acting as a canary with a policy that forbids updates to both failure domains at the same time.
2. **Geographically** separated domains provide isolation for natural disasters.
3. Using different **software versions** in distinct domains reduces the chances of a single bug breaking all servers or corrupting all data.
4. A failure domain may fail if even just one critical components fails, so using alternative components minimizes the probability that all domains fail due to a single flawed critical component.
5. High capacity components serve users and absorb spikes in requests or resource consumption due to new features.
6. The risk of a failure cam be mitigated by maintaining high availability copies with a probably lower probability of outages because of fewer dependencies and a limited rate of changes, e.g.
    i. data **cached** on local storage rather than in a remote one
    ii. **older** code and configs to avoid recent bugs in newer

All Google strategies are based on the fact that they have a huge number of resources all around the world. They *neglect* **hardware faults**, because they can easily fix by migrating software from a physical machine to another, making hardware failures not so critical.

Amazon's approach instead is focuses also on physical aspects.

A feature of almost any cloud application is the adoption of **multiple copies** of resources, each managed by a distinct server and with a **load balancer** that can also change the number of copies.

Chapter 23

Challenge III

23.1 Outline

1. A user can launch an instance of a compiler C
2. When launching the instance, the user specifies the name of the source file
3. When the compilation is over, the user specifies the name of the file to store the output code
4. The compiler updates a file log with the accounting information so that users pay for using the compiler

	Compiler	`input` file	`output` file	`log` file
User	`execute`	`read`	`write`	
Compiler				`write`
Instance		`read`	`write`	`write`

Table 23.1: Outline schema

What happens if the user transmits as the name of the output file the one of the logfile?

23.2 In-class discussion

The key issue according to prof. Baiardi is that the article asserts that ACLs and capabilities are not equivalent, because ACLs do not allow to prevent the attack while capabilities can.

Solve the problem using ACLs. "There is a *very very very* simple solution."

A simple idea is to not grant the `write` right to the compiler but the `append` right, leading the compiler to simply append to the log file both the output of the compilation and the actual log. This is not a proper solution, but still is a nice countermeasure.

	Compiler	`input` file	`output` file	`log` file
User	`execute`	`read`	`write`	
Compiler				`append`
Instance		`read`	`write`	`append`

Table 23.2: Outline schema

23.3 Proposed Ideas

I gave a quick look at this *"ACLs don't"* article, it may be the one cited by the professor when presenting the challenge.

The article proposes a sligthly different ACM than the one outlined in class 23.1.

	Compiler	`input` file	`output` file	`log` file
User	`execute`	`read`	`write`	
Compiler		`read`	`write`	`write`
Instance		`read`	`write`	`write`

Table 23.3: Article's ACM

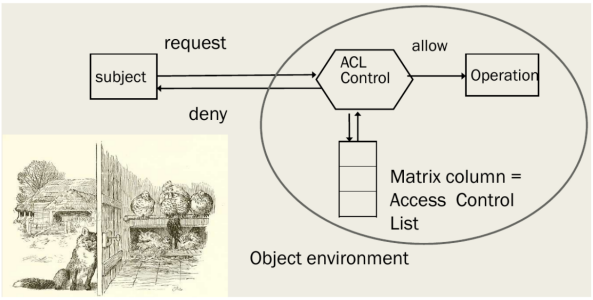


Figure 23.1: ACL schema

Recalling the ACL schema 23.1 presented in class, article's ACM 23.3 makes sense: since it's the compiler (i.e. *subject* in Fig. 23.1) which *requests* to read the `input` file to write onto the `output` file the result of the compilation, the lookup in the ACM will be performed on the *Compiler* row, not the *User*'s one.
To summarize the point made in the article, the user requests to execute `compiler("input.c","a.out")`, and after such operation is granted, it will be the `compiler` to request permission for further operations, i.e. reading input, writing output and, lastly, logging.
in case a malicious user invokes `compiler("input.c","log.json")`, when the compiler requests to write the compilation output to `"log.json"`, the operation —since the compiler has write right on `"log.json"`— is allowed, resulting in the log file to be overwritten (**"confused deputy"** problem).

I think that here lies the key issue: if the check in the ACM is done on the User's row the compiler does *not* need write access to output file and thus cannot be *"confused"*.

The compiler should write the compilation result in a "working" directory that a User can only read, and then it should be the User to request to read such compilation result and then to write it to desired output, forcing the ACM check to be performed on the User's row, not the compiler one.

	Compiler	`input` file	`output` file	`log` file	working dir
User	`execute`	`read`	`write`		`read`
Compiler	`read`			`append`	`write`
Instance		`read`	`write`	`append`	`write`

Table 23.4: Proposed ACM

23.4 Alternative Solution - Decomposition

A different approach, which instead requires more consistent changes in the software architecture, would imply decomposing the compiler in two components/modules:

1. *Compiler* producing the output file, and writing log records in a temporary `tmp` file[1].
   A pipe, a socket or a generic temporary file the writeable by *Compiler* and readable by *Logger*
2. *Logger* which receives (and validates) info to be logged from the *Compiler*

In this way we can define the security policy according to the **Least Privilege** principle, negating the `log` file `write` right to the *compiler* and granting `append` only to the *logger*, resulting in the following ACM:

	Compiler	`input` file	`output` file	`tmp` file	`log` file	working dir
User	`execute`					
Compiler		`read`	`write`	`write`		`write`
Logger				`read`	`append`	
Instance		`read`	`write`	`write`	`append`	`write`

Table 23.5: Decomposition ACM

[1] distinct from output file

Chapter 24

Internet of Things

24.1 Inside IoT

24.1.1 Hardware components

IoT devices contain **sensors**, **actuators**, or *both*.

- ◇ **Sensors** ⟶ *acquire data*
  Monitor Things and provide data about the Thing, like temperature, light intensity, or battery level.
- ◇ **Actuators** ⟶ *control/act on data*
  Control Things through hardware in the device; they represent the physical interface to the Thing that "make it go"
  e.g. controls in a smart thermostat, dimmer switch in a smart light bulb, or motors in a robotic vacuum cleaner.

All IoT devices have a way to process sensor data, store –if necessary– such data locally, and provide the computing power that makes the device operate. The data **processing component** of the IoT device coordinates data from multiple sensors or stores in flash memory.

24.1.2 Firmware

The **Firmware** is the onboard software that runs an IoT device sits between the hardware and the outside world.

For simpler resource-constrained devices the firmware may be **embedded**. More advanced devises instead nowdays have an entire **OS** as firmware, providing an *abstraction layer* between the hardware and other software on the device.

A popular OS choice is **Busybox**

24.2 IoT Attacks

Security is usually an *afterthought* in IoT because it is difficult to create a cheap reliable, resource-constrained device that can connect to a wireless network, having very little power consumption.

Attack Vectors

1. **Weak Passwords**
   To simplify the device setup and use, the manufacturer offers typically weak login credentials
2. **Lack of encryption**
   Many IoT devices do *not* support encryption, to save battery and performance.
3. **Backdoor**
   Is common practice for manufacturers to put "hidden" access mechanisms to simplify the support. Once a backdoor becomes known, the manufacturer can either remove it, or make it more difficult to be accessed (or so they think ☺)
4. **Internet Exposure**
   Unlike a hardened server where you can control the firewall and how the host is accessed, most IoT devices have little or no security and accept most internet traffic, making them susceptible to attack.

Countermeasures

1. Always **change** the *default password*
2. **Remove** devices with *telnet backdoors*
    i. To discover such devices you can use IoT device scanners that check with **Shodan**, an IoT search engine, to reveal if your devices are vulnerable based on the IP address of the scanning computer.
3. **Never expose** a device directly to the internet
    i. When you consider whether or not to expose a device to the internet by opening up your firewall, the right answer is almost **always no**.
4. IoT device scanner can run a *"deep scan"* to check for any open ports on your publicly exposed IP address assigned by your ISP.
    i. **Port Scan** all your machines

24.2.1 Shodan

Shodan is a *search engine* for finding specific devices, and device types, that exist online.
The most popular searches are for things like webcam, linksys, cisco, netgear, SCADA, Default password.

It uses a specialized scanner to scan the **entire Internet** and parse the banners that are returned by various devices.

Shodan can tell things like what web server –and version– is most popular, or how many anonymous FTP servers exist in a particular location, and what make and model the device may be.

You start by navigating to the main page, and then entering into the search field, like you would any other search engine.

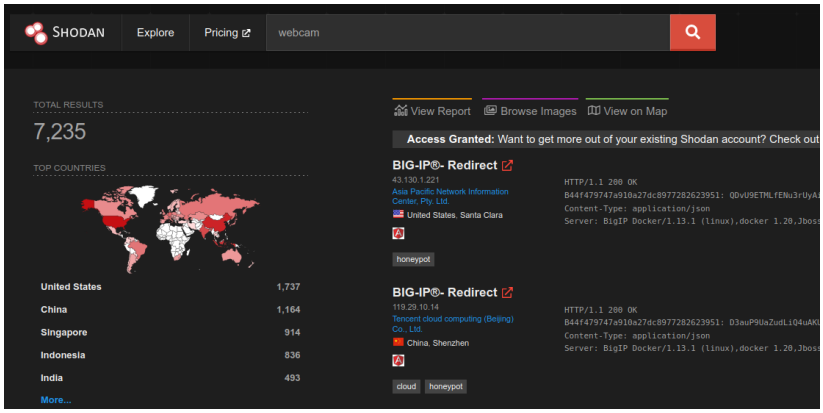


Figure 24.1: Shodan search screenshot

24.3 Device Classification

IoT Nodes may be classified by performance or functionally

Performance

1. **Ultra-constrained** node
   RTOS or bare metal with 16K of RAM. Energy harvesting limits radio transmissions to conserve power.
2. **Constrained** node
   RTOS with 32K-64K RAM. Most likely running on a battery and software optimised for battery life. Again minimizing radio transmission
3. **Mainstream** node
   A feature rich RTOS with 128K RAM. More complex interaction with the context since there is room for more local operation, as opposed to sending all data upstream.
4. **Gateway** node
   An advanced OS with 64MB RAM. A sophisticated node with advanced software and runs from main power. Multiple radios to support the local network.

Functional

1. **Simple node**
   Not aware of the rest of the local network. It collects and reports information to the specified destination. Any of previous 1-3 may be a simple node it depends upon interactions.
2. **Smart node**
   fully aware of all other network nodes mainly via software that understands mesh networks, local topologies and authorised interactions between nodes in the same network.
3. **Access node**
   the edge box to connect the local network to the Internet via whatever broadband link is appropriate for the application. It has multiple radios facing the local network.
4. No restrictions on creating a node that is a smart node and acts as a gateway.

24.3.1 Architecture

Software productivity requires 32 bit real time embedded processor to quickly develop and release software to handle connectivity, security and IoT applications while managing and controlling multiple sensors, and since one node design may be used to release products for multiple different markets, software productivity is essential.

To optimize power consumption multiple processors may be required, each for a different task:

1. handling radio stacks and connectivity,
2. managing the sensors and actuators
3. running system and network stack

No need to wake up the main processor if the radio one may check for incoming messages

For what concerns **memory**, most embedded systems include:

- ◇ *Flash* memory to store the program,
- ◇ *SRAM* to store code and data
- ◇ *ROM* to hold the basic system description

24.4 Security in IoT

Bruce Schneier thinks that security even in a world where *everything is connected*[1] can be addressed as if there's *"nothing new"* from standard ICT security processes.

1. Most software is poorly written 2. Internet was never been designed to be secure 3. Extensibility is a problem = A computer is an universal Turing machine = If it is smart is vulnerable 4. Complexity helps the attacker 5. New vulnerabilities to interconnection 6. Attacks always get faster, smarter and cheaper

What changes does IoT world introduce?

1. **Automation**, **autonomy**, **physical** agents
2. **Patching** becomes much more complex or impossible and patches should be available for all the life of the things not of the computer (you cannot replace the computer but the thing)
3. The same vulnerability and the same attack work both in ICT and internet of things
4. **Integrity** and **availability** are more important than confidentiality
   leaking health records vs hacking a body device

Schneier is not completely wrong, in fact most "old" vulnerabilities can strike also IoT devices, but there are three key points that are not negligible:

1. the **physical** nature of IoT
2. the **number** of devices
3. the existence of **preloaded code**

There are two approaches to address the security of an IoT device:

1. *Attack the device*
   Any smart device increases your system attack surface and its connection can result in new attacks, making the device an intermediate step aka *"stepping stone"*

[1] (~~connected car~~ ⟶ *computer with four wheels*, ~~connected refrigerator~~ ⟶ *computer that keeps things cold*)

2. *Attack **from** the device*
   Any smart device can store some malware to attack your system, making attacks from the device possible even from simple smart nodes.

   While most of security research has focused on the first issue, the *second* one is becoming more and more critical.

In general, note that IoT includes a huge number of devices with code that cannot be easily accessed, tested or patched.
According to **Baiardi**, **fuzzing** and **reverse engineering** will become more and more important to analyze devices and preserve *confidentiality* and *integrity*.

24.5 New perspective on Attacks

The interaction of an IoT device with the physical world enables to *exploit physical properties* to attack an ICT system in new ways. An interesting example are attacks against a voice interface using frequency that the interface can hear but the human user cannot hear, also known as **DolphinAttack**s.

The attack is effective on popular speech recognition systems, e.g. Siri, Cortana and Alexa.

Countermeasures can either be hardware, e.g.microphone enhancements to drop frequencies above 20Khz, or software, e.g. classifying the signal to discover modulated voice.

In late 2016 the *Mirai botnet* launched $15k$ DDoS attacks, exploiting as nodes mainly embedded and IoT devices which exposed `SSH/Telnet` services; the attack infrastructure attempted to login to such devices[2] using a set of **46** hardcoded passwords, some traceable to a vendor, and then wait for commands from a C2 server.
A couple of interesting aspects of Mirai are the hardcoded list of IPs–composed of adddresses belonging to *US Postal Service*, *Dept. of Defense*, `IANA`, *HP* and *General Electric*–which bots avoided when performing scans, and its *"territorial"* nature: bots included scripts whose purpose was to eradicate other worms an trojans as well as prohibiting remote connections to the hijacked device.

A proposed taxonomy on attacks to IoT devices is based on how the attacker *deviates* features from their "official" functionality, resulting in 4 different categories:

1. **Ignoring** the functionality
   The attacker *ignores* any physical functionality of the IoT device, and considers it only as a standard connected computing device; Since most IoT devices are cheap and with minimal security protections, without the possibility to be upgraded or patched, they are the perfect target.
   These attacks are a serious threat but the least interesting ones because they are applicable to any networked device and are not unique to IoT devices. Nothing new so far, but huge *#devices*
2. **Reducing** the functionality
   The attacker tries to kill or limit the designed functionality of the IoT device, e.g. the TV/refrigerator stops working. Seems dull, but consider wide organizations or *connected medical devices*: attacks on such devices may be fatal.
   An attacker may use **ransomware** to temporarily lock an *expensive physical device* and demand a large payment to restore its functionality.
3. **Misusing** the functionality
   These attacks use –rather than disabling– a functionality of the physical device, but in an incorrect or an unauthorized way, usually resulting in annoying pranks that do not violate integrity; e.g.the attacker turns on heaters in august.
4. **Extending** the functionality
   Extending a functionality requires imagination and sophistication, but may result in harmful unexpected effects, like a Roomba opening the lock of a door.
   A concrete example exploited LED dimmering to transmit information.

24.6 Privacy Concerns

When an IoT device is plugged into a home network it will start collecting data and doing its job sending data "home" to the companies servers, to provide the "smart" features. It is not trivial to understand which data is **collected** who is **responsible** for such data. Even if the timing of opening/closing the fridge seems useless, the synergy and interconnection in the massive data going outside your home network, may be used to infer important information, e.g. the presence or absence of people in a house.

[2] IP cameras, DVRs, consumer routers

24.7 IoT Best Practices

On the 2^{nd} of November 2019, the *IoT Security Foundation* published a few guidelines regarding security for IoT systems. Some of them are already discussed in the previous section, while in the following subsections we will address others.

24.7.1 Physical Security

Any interface used for administration or testing should be (*physically*) removed from the production device. The device should also make its circuitry physically *inaccessible* to avoid **tampering**, and in general should be –along with its packaging– **tamper-evident**.

24.7.2 Secure Boot

Firstly it is advised to always use the ROM-based **secure boot** function, generally implying a multistage bootloader initiated by –a minimal amount of– **read-only code** stored in a *one-time programmable* memory.
Besides, a hardware-based tamper-resistant capability should be used to run the trusted authentication/cryptographic required for the boot process.

There are some attacks in booting called "*Time of Check* to *Time of Use*", which can be strongly limited by **validating** boot code immediately before use at *each boot stage*.

Lastly, any *failure* in the boot sequence should **fail gracefully** in a safe and secure state.

24.7.3 LogoFAIL

A LogoFAIL attack consists three steps:

1. The attacker prepares a malicious logo image and stores it into the ESP or inside an unsigned section of a firmware update, and it restarts the device.
   No phyisical access is required for this step. It can be performed by exploiting unpatched vulnerabilities in browsers, media-players, etc. or by gaining brief access to a device while it is unlocked.
2. During the boot process, vulnerable firmware will load the malicious logo from the ESP and parse it with a vulnerable image parser, thus the attacker can hijack the execution flow by exploiting a vulnerability in the parser itself.
3. The attacker achieves arbitrary code execution during the DXE[3] phase, which means complete game-over for platform security.

Note that malicious code *never reaches* the hard drive, and that once the hijacked image is in place, reinstalling the OS or replacing the main drive, won't remove it, ensuring that the device remains infected.

Attacks, such as LogoFAIL, starting from the firmware level can be leveraged to install a bootkit and subvert any OS-level security mechanism, while remaining completely *undetectable* by security detection solutions. Modern "below-the-OS" such as *Secure Boot* are useles against this threat.
Note also that LogoFAIL, since it targets UEFI code and does not depend on a specific architecture, represents a **cross-silicon exploitation**.

LogoFAIL was discovered by researchers using **Fuzzing**: *"When the campaign finished, we were overwhelmed by the amount of crashes we found, so much that triaging them manually was quite complicated"* the researchers wrote. In all, they identified 24 unique root causes, 13 of which they believe were exploitable.

The results raised a vexing question:

If fuzzers identified so many exploitable vulnerabilities, why the developers of and the OEMs selling the devices hadn't already used these tools and fixed the underlying bugs?

Fuzzing is fundamental according to Baiardi

24.7.4 Secure OS

The OS should include only the components (libraries, modules, packages, etc.) required to provide the functions of the IoT device.

[3] Driver Execution Environment

The devices should be shipped with the latest stable OS release and should be configured with the most secure configuration available, e.g. disabling unused ports, protocols and services; OS components should be kept updated throughout time.

The *Least Privilege Principle* should be strictly applied to all directories and files, and the root file system should not be writeable by users and applications.

24.7.5 Credential Management

Every device should be uniquely **identifiable** by means of a factory-set *tamper-resistant hardware identifier*.

Only non-trivial **passwords** should be allowed, and they should be managed and stored properly using *industry standard* **encryption** mechanisms.

Also **digital certificates** should be handled carefully, and a single certificate should not be used to identify more than one device.

24.7.6 Secure Software Update

Packages of new releases should be **encrypted** to hinder reverse engineering, and the updated routine should always validate integrity and authenticity of a package update before the installation begins.

As for *Secure Booting*, a **fail-safe** mechanism must be provided when performing updates.

24.7.7 Side Channel Attack

A **Side Channel** is an unintended/unanticipated capability to observe changes in the state of a system, where system could be at the chip, board, application, device or network level.

Side Channel Attacks deduce information based on these changes and then use that information to exploit the system. For example monitoring variations in temperature, timing of certain events, changes in power consumption, emissions of sound, electromagnetic radiation at any frequency etc. can *leak information* about the system from which data may be inferred or deduced.

Monitoring of a system's behavioural symptoms may be further augmented by using **Fault Injection**, i.e. deliberately running a system under conditions outside those for which it was designed, possibly allowing to establish side channels *not* available under normal operation.

Chapter 25

Stuxnet

In the summer of 2010 **Stuxnet**, a malware of unprecedented complexity made the news, by exploiting **multiple** (3) zero-day exploits.

Stuxnet didn't act like any previous malware before and can be considered pioneristic from many points of view, but mostly because its objective was **not** the theft or manipulation of data, but the ***physical destruction*** *of gas centrifuges* in the Natanz fuel enrichment plant, the crown jewel of Iran's nuclear program.
So, Stuxnet was not implementing a strict ICT attack, because its purpose was to cause a physical effect.

Does IoT ring a bell?

In ICS most automated factory processes are ran by `PLC`s (Programmable Logic Controllers), which can be programmed (typically from Windows) using basic scripting or GUI-aided logic. However, they typically are not connected to the internet, thus creating an *"air-gap"*.
Stuxnet targetted a specific PLC control system, `SIMATIC PCS 7 Process Control System`, programmed using WinCC/STEP 7.

Stuxnet distributed mostly through USB keys, bypassing the air gap.

The attack was never reclaimed by any group;even though there were some hidden clues in the source code, none of them were consistent, besides

"Symantec cautions readers on drawing any attribution conclusions. Attackers would have a natural desire to implicate another party"

Chapter 26

Secure OS

26.1 TrustZone Hardware

The **TrustZone** hardware architecture aims to provide a security framework to enables a device to counter multiple threats, pursuing the following goal:

enable the construction of a programmable environment to protect from attacks to confidentiality and integrity of almost any asset

This architecture splits –both HW and SW– components into two categories, exploiting a strong hardware logic to ensure a well defined security perimeter between the two:

1. **Secure world** - for the security subsystem
2. **Normal world** - for everything else

Releavant features of the TrustZone architecture include:

- ◇ Two virtual cores for each physical CPU core, one for *Secure* world and the other for *Normal* world, along with a robust **context switch**; this system allows sharing a physical core between the two worlds in a *time-sliced* fashion.
  - During the context switch the core goes through a *monitor mode*, whose entry points are tightly defined, especially the ones from the *Normal* world.
    The *monitor* mode's purpose is to provide a robust gatekeeper between the two worlds.
  - For each virtual core there is a separate MMU, and each world manages its own translation tables which include the `NS` bit —discussed in the following points—.
    `TLB`s are tagged with the identity of the world which performed the walk, allowing for both world's entries to co-exist.
- ◇ Security-aware *debug infrastructure* which provides **control** –instead of *access*– to *Secure* world debug.
- ◇ `NS` *Non-secure* bit for each `read`/`write` operation on the main system bus.
  Every bus master sets the `NS` bit at the hardware level, inheriting it from the identity of the *virtual CPU core* requesting the operation, making it impossible for *Non-secure* masters to access *Secure* slaves.

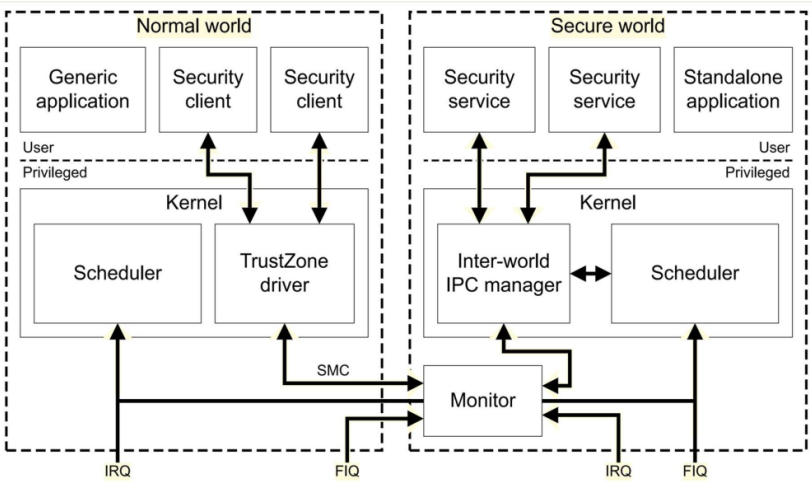


Figure 26.1: *TrustZone* - Secure OS

26.2 Rogue Firmware

An erroneous update or because of a malicious software developer, a device can store some malicious software that attacks the system or offers an hidden backdoor.

Some run time attacks can be discovered by memory introspection or attestation, other ones can be discovered by an *host-based* Intrusion Detection System called **Doppelganger**.

A Doppelganger first analyzes the firmware of the *embedded device* to detect live code regions[1] therein, where it randomly inserts its **symbiotes** (*watchpoints*), and computes a CRC32 checksum of such randomly selected regions.

During the firmware execution, every time the *symbiote manager*[2] detects a symbiote in memory,

1. it stops the execution process (symbiote=breakpoint),
2. compares the current memory area checksum of the symbiote one,
3. Dopplganger considers a mismatch an evidence of a modification attack and it prevents the processor from running the code.

Doppelganger does not defend against attacks that load code in the dynamic memory

26.3 Secure Management of a Device

The manifacturer must flash bootstrap credentials onto the device, which allow it to reach its "home" bootstrap sever. Then it can ask such bootstrap server for a DM server address and the respective credentials, to allow the DM server to manage the device.

26.4 Encryption in IoT

Simon and **Speck** are two families of block ciphers proposed by NSA, with `SIMON` being tuned for optimal hardware performance, while `SPECK` for software.

Depending on the security level requested, they have different implementations varying block sizes: e.g. `SIMON 64/128` refers to a `SIMON` variant of the cipher with a block size of `64 bits` and a key size of `128 bits`.

Their aim are *generality* and *simplicity*, to hopefully provide high performance even on future constrained platforms, whatever they may be. Currently they can exploit also the capabilities of specific-purpose ASIC/FPGA boards; furthermore, they provide low latency, ease of protection against side-channel attacks, and are efficient from a power consumption point-of-view, making them a perfect fit for the IoT device industry.

Sadly, since NSA isn't completely trusted by many countries, `SIMON` and `SPECK` have not yet become an industry standard, even if they have been tested by both NSA and academical researchers since 2014.

[1] executable parts of the firmware
[2] Software that the Doppelganger adds *before* the firmware

Chapter 27

Artificial Intelligence

Cybersecurity attacks targeting AI have a **different nature** from the ones studied before, since vulnerabilities in AI cannot simply be identified and patched, they are more *intrinsic*: the algorithms themselves and their reliance on data are the problem.

Despite this fundamental difference, the two are *linked* in important ways: in fact, many AI attacks are aided by gaining **access to assets** such as datasets or model details. **Confidentiality** attacks enable adversaries to obtain the assets needed to engineer *input* attacks, while **integrity** attacks will enable adversaries to make the changes to a dataset or model needed to execute a *poisoning* attack.
More generally, while AI attacks can certainly be crafted without accompanying cyberattacks, strong traditional cyber defenses will increase the difficulty of crafting certain attacks.

Since AI attacks are becoming popular only in the last few years, it is still unclear whether they can be sufficiently practical and effective to represent a relevant threat.
However, even if some may say that AI attacks do not deserve equal consideration with their traditional cybersecurity attack counterparts, such point of view is **incorrect**.

Most AI systems are **opaque**, meaning that the systems accepts inputs, and generates outputs without ever revealing the internal logic, algorithms or parameters. Also training data sets, which effectively contain all the knowledge of the trained system, are usually kept **confidential**.
This makes it usually impossible for an outside observer to apply reverse engineering techniques to determine exactly how a system works, or why it produces particular outputs.

27.1 Attack categorization

27.1.1 Poisoning

1. **Dataset** poisoning is the most direct attack:
   Introducing during data *collection* or *curation* incorrect, or incorrectly labelled, data into the set, may result in the entire learning to be disrupted.
   i. **Label Poisoning** (*Backdoor* Poisoning)
   Adversaries inject mislabeled or malicious data during the training phase to influence the model's behavior during inference, eventually resulting in triggering an unexpected output by a designed input (called *triggers*).
   ii. **Training Data Poisoning**
   Adversaries modify a significant portion of the training data to influence the AI model's learning process. The misleading or malicious examples allow the attacker to *bias* the model's decision-making towards a particular outcome.
2. **Algorithm** poisoning occur when an attacker interferes with the learning algorithms.
3. **Model** poisoning occurs when the entire deployed model is simply replaced by an alternative one.
   This is similar to a traditional attack where the electronic files comprising the model could be altered or replaced.

27.1.2 Input/Evasion attacks

In an **input attack** (aka *evasion attack*) an attacker modifies the AI system input to cause the system to malfunction, acting on the *communication channel* or directly on the *input source*, which is the most interesting case.

Data perturbations can be very small, making them very hard, if not impossible, to detect.

There is a famous example which highlighted how by changing just a few pixels of an input, the system might be forced to wrongly identify an image.

27.1.3 Microsoft "Failures" categorization

Intentional failures

- ◇ **Perturbation** attack
  Attacker modifies the query to get appropriate response
  - **Image**: Noise is added to an X-ray image, which makes the predictions go from normal scan to abnormal.
  - **Text translation**: Specific characters are manipulated to result in incorrect translation. The attack can suppress specific word or can even remove the word completely
  - **Speech**: Researchers showed how given a speech waveform, another waveform can be exactly replicated transcribes into a totally different text
- ◇ **Poisoning** attack
  Attacker contaminates the training phase of ML systems to get intended result
  - **Targeted**: the attacker wants to misclassify specific examples
  - **Indiscriminate**: the aim here is to cause DoS like effect, which makes the system unavailable.
    In a medical dataset where the goal is to predict the dosage of anticoagulant drug Warfarin using demographic information, etc. Researchers introduced malicious samples at 8% poisoning rate, which changed dosage by 75.06% for half of patients.

    In the Tay chatbot, future conversations were tainted because a fraction of the past conversations were used to train the system via feedback.
- ◇ **Model Inversion**
  Attacker recovers the secret features used in the model by through careful queries
- ◇ **Membership Inference**
  Attacker can infer if a given data record was part of the model's training dataset or not
- ◇ **Model Stealing**
  Attacker is able to recover the model through carefully-crafted queries
- ◇ **Reprogramming ML system**
  Repurpose the ML system to perform an activity it was not programmed for
- ◇ **Adversarial Example in Physical Domain**
  Attacker brings adversarial examples into physical domain to subvertML system
  e.g: 3d printing special eyewear to fool facial recognition system
- ◇ **Malicious ML provider recovering training data**
  Malicious ML provider can query the model used by customer and recover customer's training data
- ◇ **Attacking the ML supply chain**
  Attacker compromises the ML models as it is being downloaded for use
- ◇ **Backdoor ML**
  Malicious ML provider backdoors algorithm to activate with a specific trigger
- ◇ **Exploit Software Dependencies**
  Attacker uses traditional software exploits like buffer overflow to confuse/control ML systems

Unintentional failures

- ◇ **Reward Hacking**:
  Training mismatch with reality
- ◇ **Side Effects**
- ◇ **Distribution Shift**:
  Usage in environment different from test
- ◇ **Natural Adversarial**:
  Unexpected real data
- ◇ **Common Corruption**:
  Unable to manage perturbations
- ◇ **Incomplete Testing**

27.2 ETSI on Securing AI

European telecommunication Standard Institute **ETSI** has produced interesting documents on AI security defining:

- ◇ A threat ontology
- ◇ An attack taxonomy
- ◇ Attack mitigation

Currently there are other efforts by Google and others but not so well focused and detailed as the ETSI standard

They also provide a classification of learning methods

- ◇ *supervised*: training data labeled
- ◇ *semi-supervised*: training data partially labeled
- ◇ *unsupervised*: training data unlabeled
- ◇ *reinforcement*: a policy defining how to act is learned by agents through experience to maximize their reward

The question of securing AI systems can be simply stated as ensuring the *confidentiality*, *integrity* and *availability* of those systems throughout their lifecycle.

Lifecycle Phase	**Issues**
Data acquisition	Integrity
Data curation	Integrity
Model design	Generic issues only
Software Build	Generic issues only
Train	Confidentiality, Integrity, Availability
Test	Availability
Deployment	Confidentiality, Integrity, Availability
Updates	Integrity, Availability

Table 27.1: AI Lifecycle and corresponding issues

Chapter 28

AI - Discovering Vulnerabilities

1. *If your data is poisoned or tampered, how would you know?*
2. *Are you training from user-supplied inputs?*
3. *If you train against online data stores, what steps do you take to ensure the security of the connection between your model and the data?*
4. ...

The real question is *"Can you discover whether your data has been "*attacked*" in some way?*

28.1 Mitigation

How can we enrich the robustness against AI data attacks, and mitigate the eventual impact.

28.1.1 Poisoning - Sanitization

A method to mitigate data **poisoning** is **RONI** (*Reject On Negative Impact*), which relies on identifying outliers by training the model with and without each (data) point and comparing the *performance*.
Due to frequently retraining its run-time **overhead** is significant, and as a result its performance is worse than later proposals.

An alternative is **TRIM**, which iteratively estimates the model parameters and trains on the subset of best-fitting input points at the same time, until convergence is reached.
It is much better than RONI ☺. However, TRIM is devised for **linear regression** only and thus *not* applicable to (deep) neural networks.

There are also **provenance-based** techniques, which use metadata about data provenance[1] to cluster data accordingly.
Due to the additional information, it achieves **better results** than RONI and is **more efficient** by the average cluster size, since the model is not retrained for each individual point but only the *cluster centroids*.
The intuition is that the probability of poison for data points of common provenance is strongly correlated.

Keyed Non-parametric Hypothesis Tests (*KNHT*) assume a set of clean training data describing intended *data distribution* and inspect newly-collected ones to compare the two sets and if their similarity is insufficient, the newly collected ones are further inspected.
KNHT does comparison after mapping the distributions into another space via functions with secret keys.

28.1.2 Backdoors and Triggering

Backdoors attacks involve two steps:

1. *Backdoor embedding*
2. *Triggering*

In **development** stage, backdoors and triggers can be detected and even removed by modifying the addressed model; in **deployment** stage instead, backdoor detection aims at revealing potential weaknesses of the addressed model and trigger detection goal is to prevent backdoors from being triggered.

[1] the model assumes such provenance metadata is correct

28.1.3 Evasion attacks