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

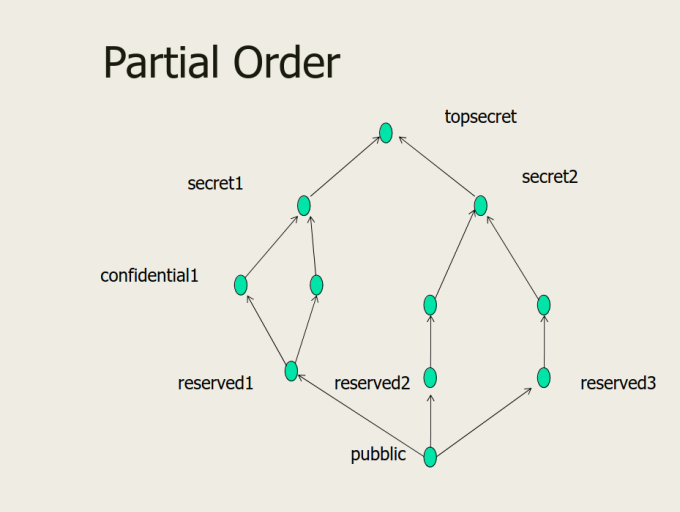
For each object there is an:

- ◇ 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).

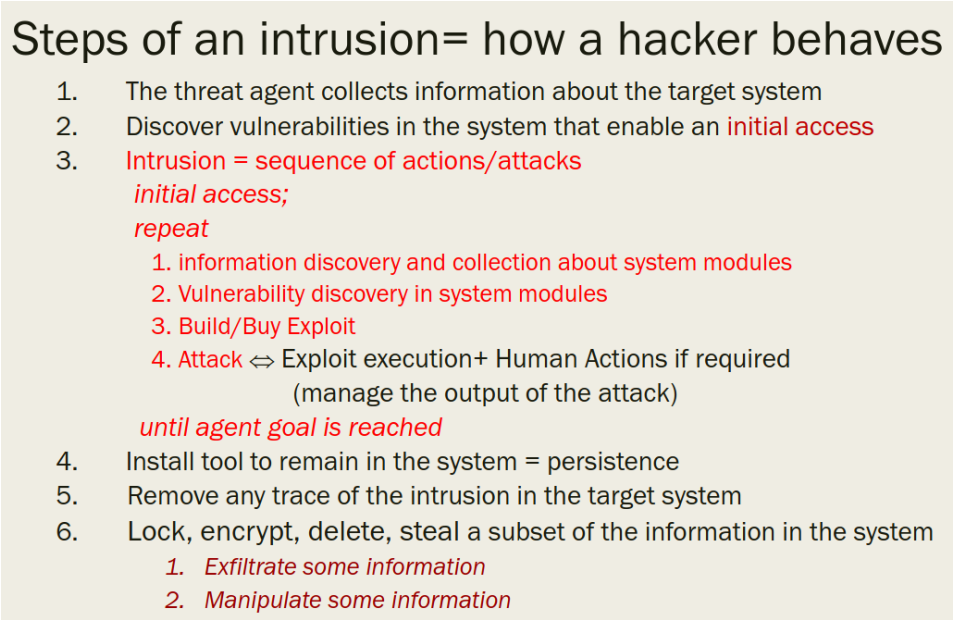
It is possible to **assess risk** only if assets, vulnerabilities and threat agents are known for a given system.

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.

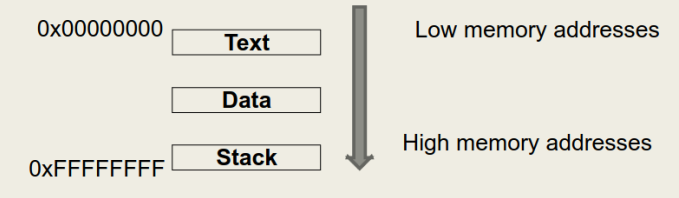
A dummy but instructive vulnerabilties classification distinguishes two major classes:

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

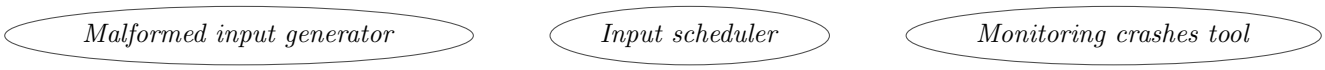
Static Application Security Testing (SAST) indicates tools of static analysis, which has advantages like scalability and easy patching, but is limited on other aspects, such as:

- ◇ 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

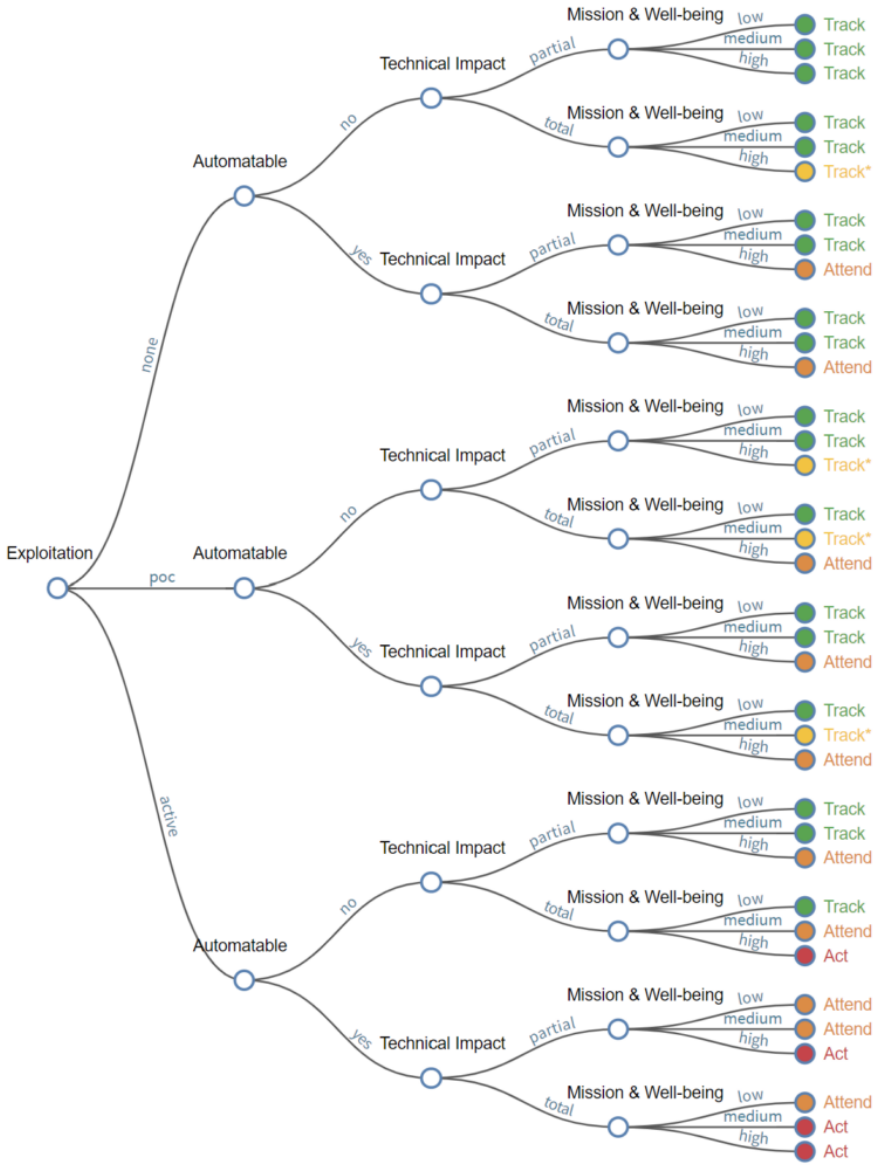


Figure 7.1: SSVC Decision tree

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.

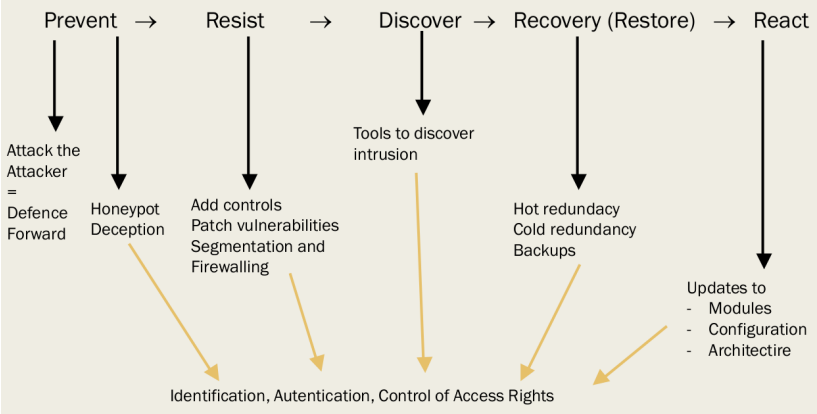


Figure 8.1: Countermeasure more detailed classification

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.

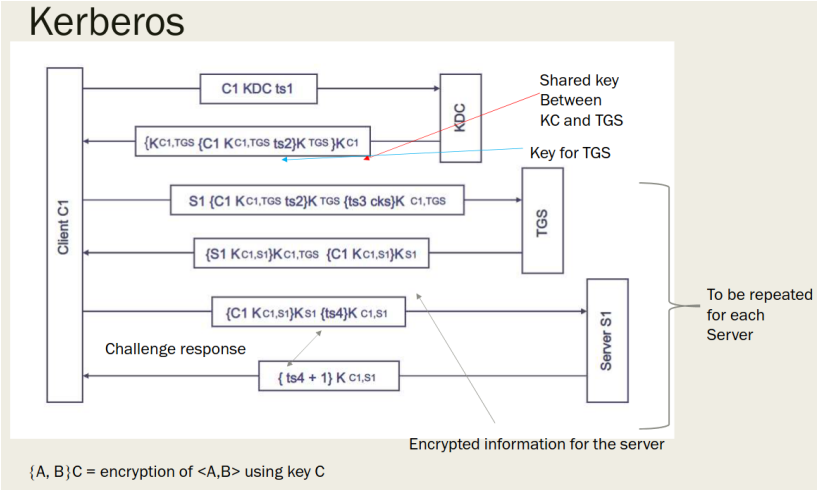


Figure 8.2: Kerberos messages exchange

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.

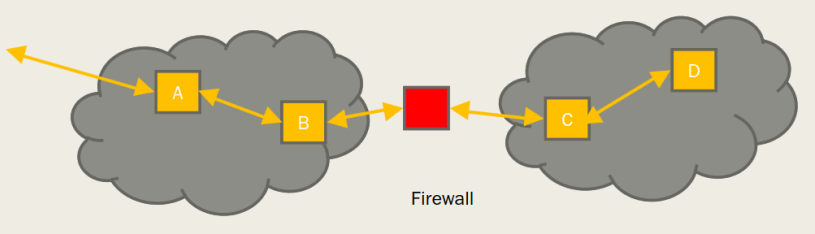


Figure 11.2: Segmented network and the need for Pivoting

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.

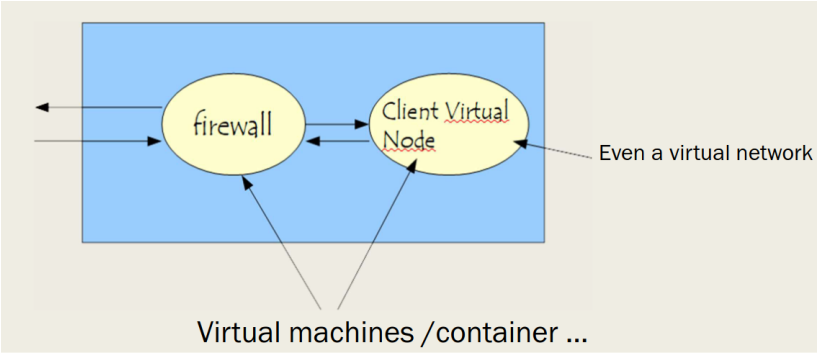


Figure 11.3: Firewalling in cloud environments

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.

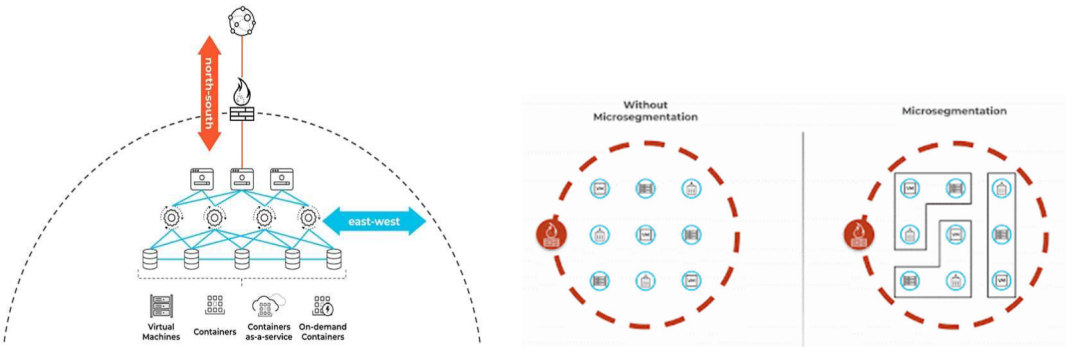


Figure 11.4: Microsegmentation example figures

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 (?)

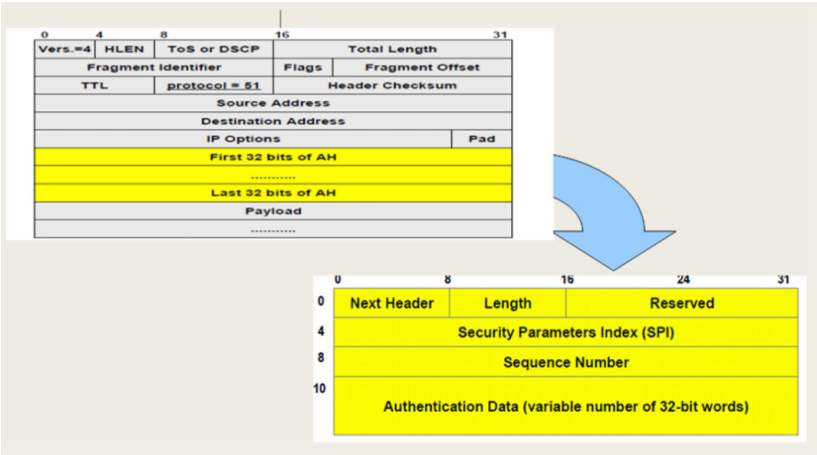


Figure 11.5: Authentication Mode header

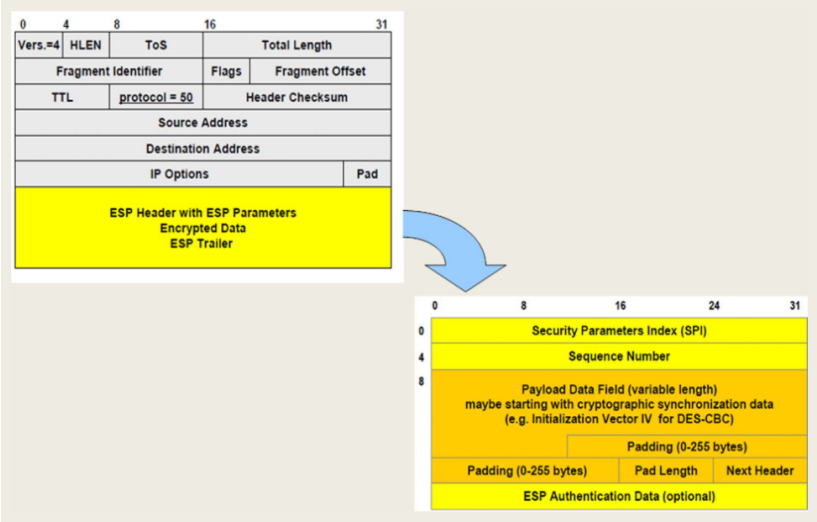


Figure 11.6: ESP mode

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

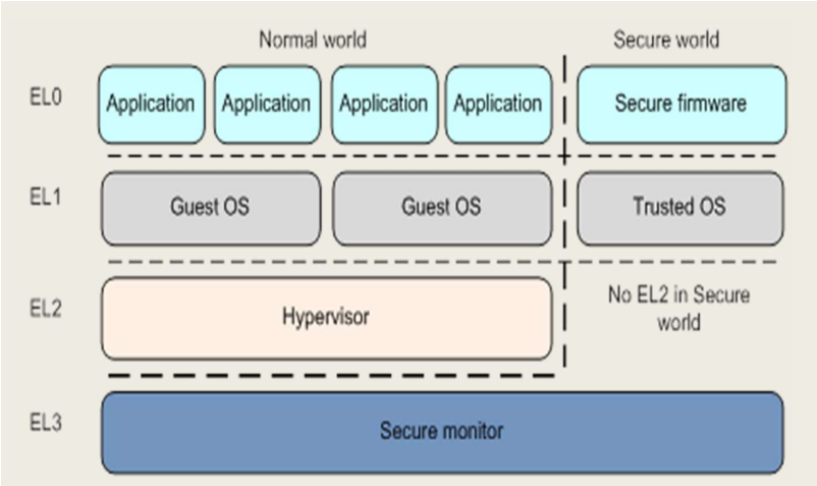


Figure 12.1: TrustZone overall architecture

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.

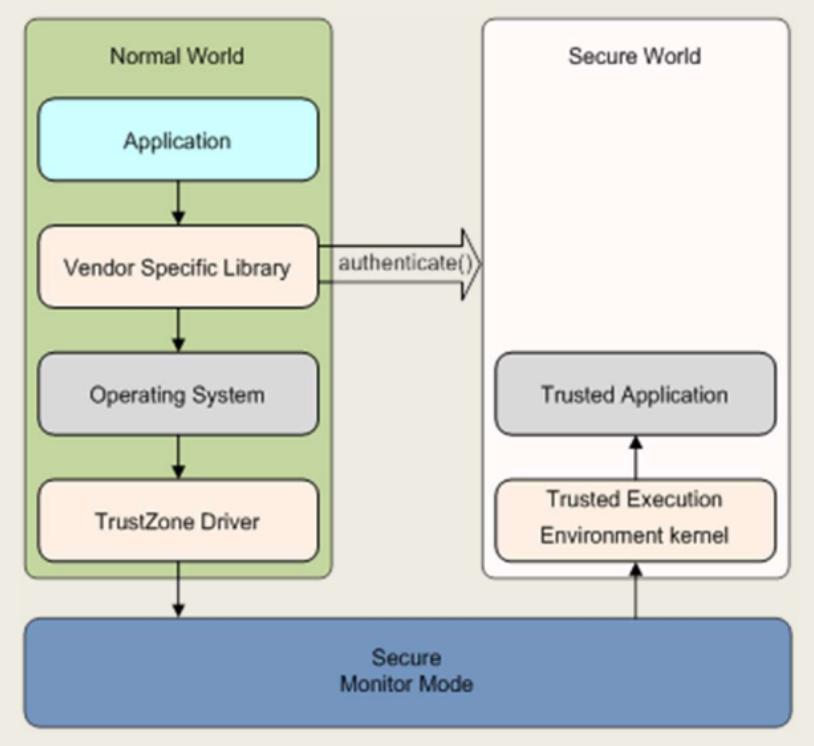


Figure 12.2: TrustZone processor architecture

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.

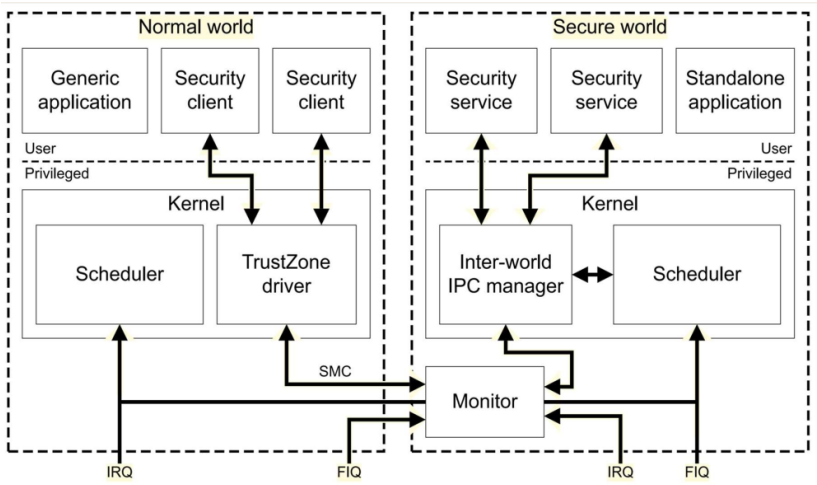


Figure 12.3: TrustZone secureOS

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

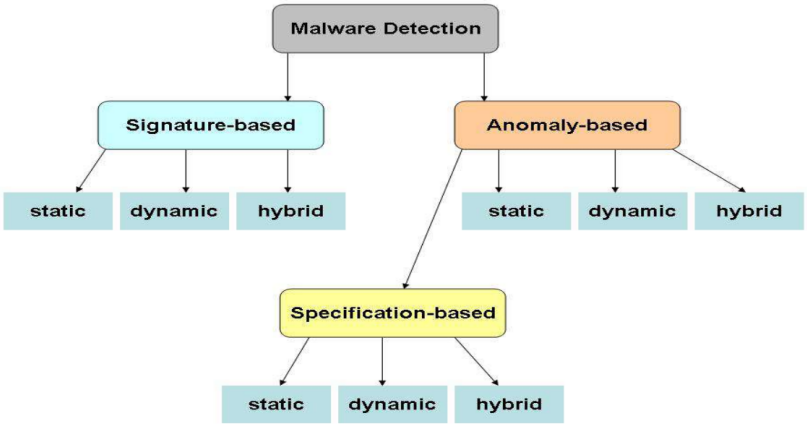


Figure 13.1: Intrusion Detection Taxonomy

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.

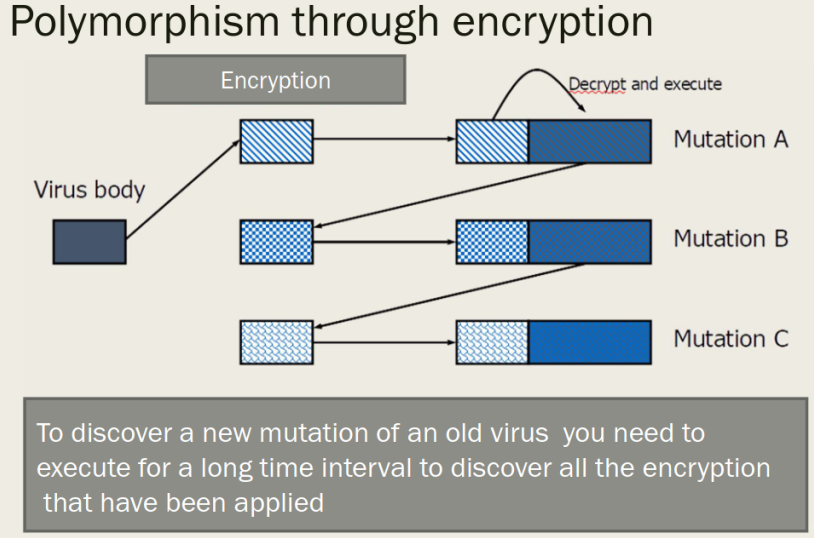


Figure 14.1: Polymorphism through encryption

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*

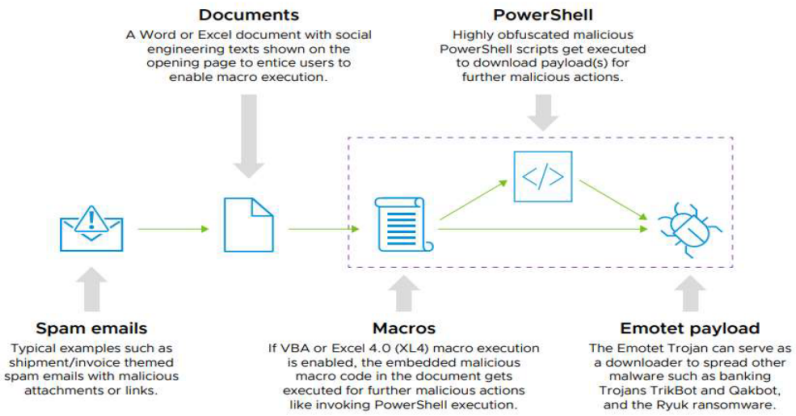


Figure 14.2: Emotet

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.

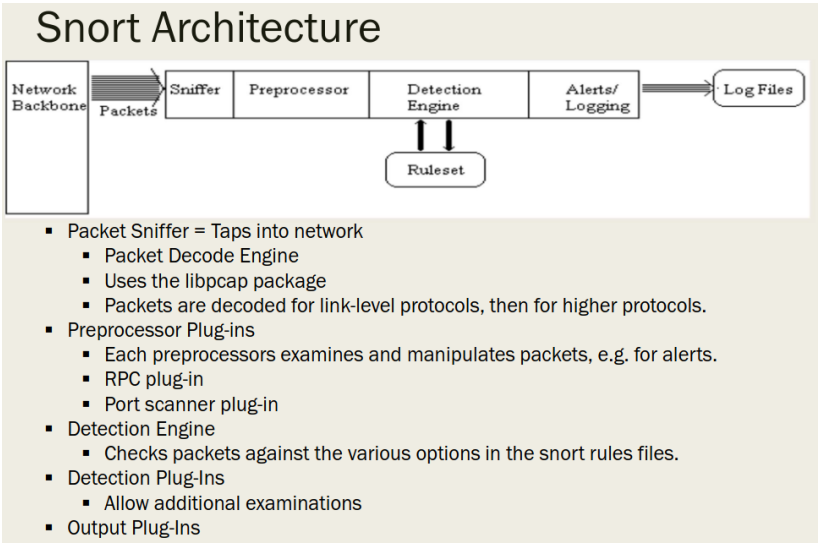


Figure 15.1: Snort architecture

[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.

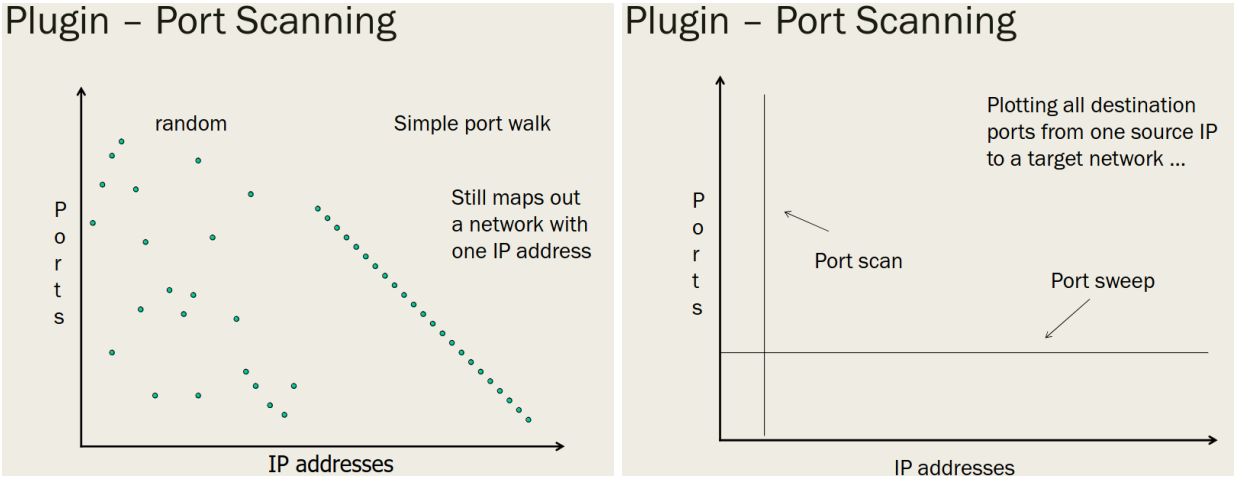


Figure 15.2: Snort port scanning plugin

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)

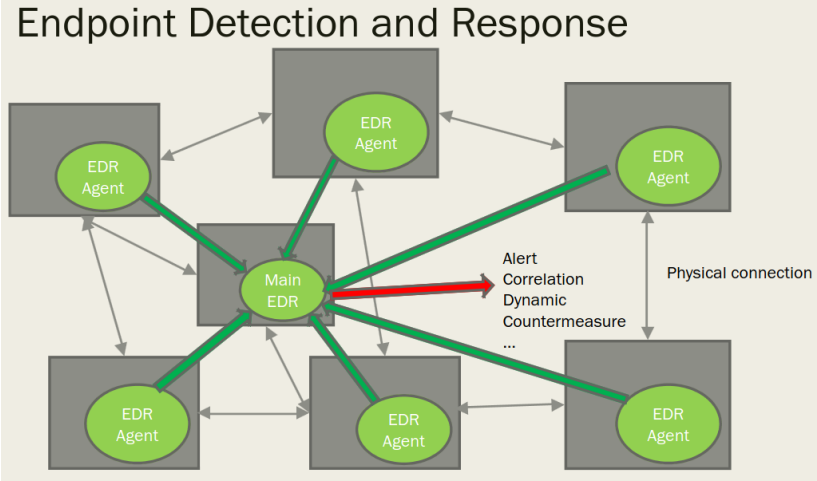


Figure 15.3: EDTR architecture

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.

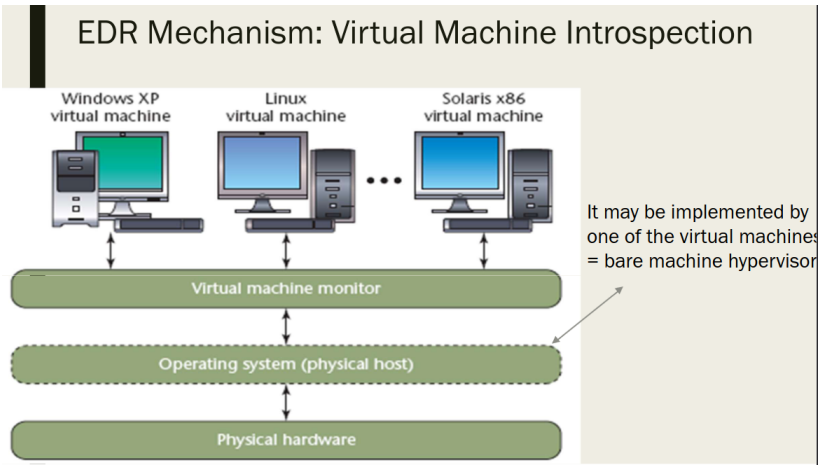


Figure 15.4: VMI arch

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.

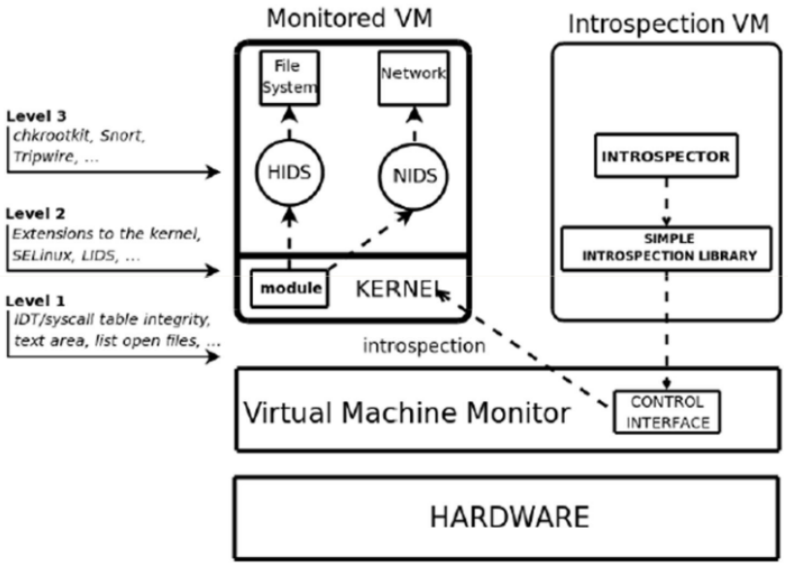


Figure 15.5: VMI Chain of Trust

Chain of trust - Intuitive

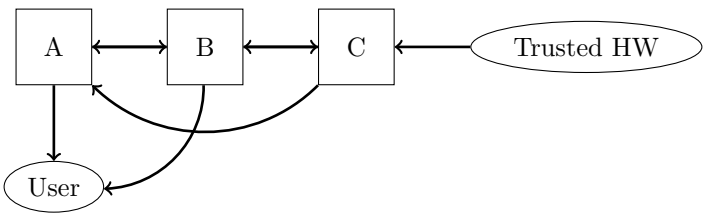


Figure 15.6: Chain of trust simplified schema

[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

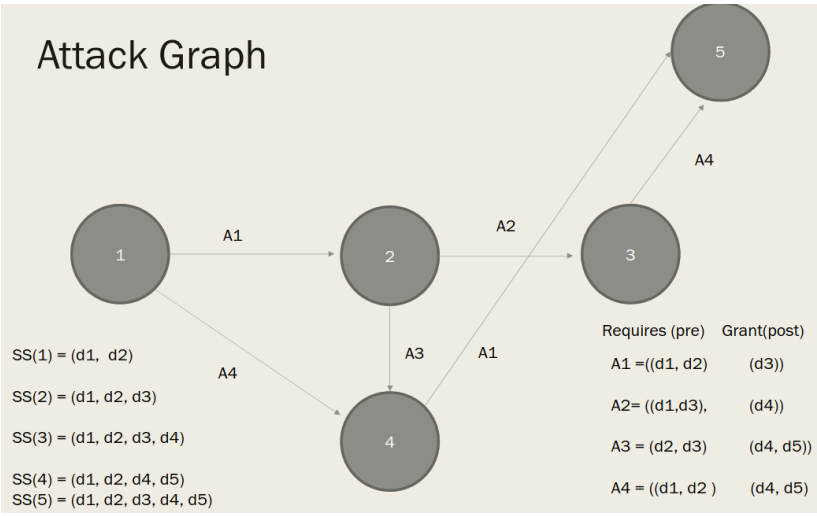


Figure 16.1: Simple attack graph example

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

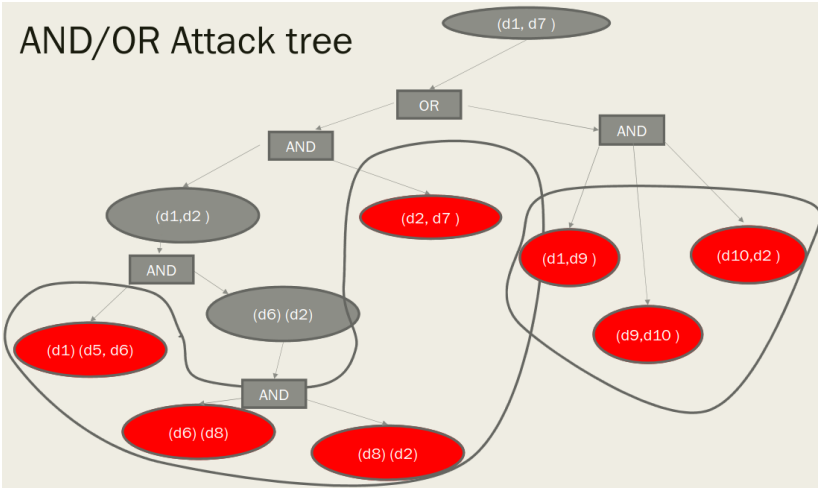


Figure 16.2: Attack tree

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.

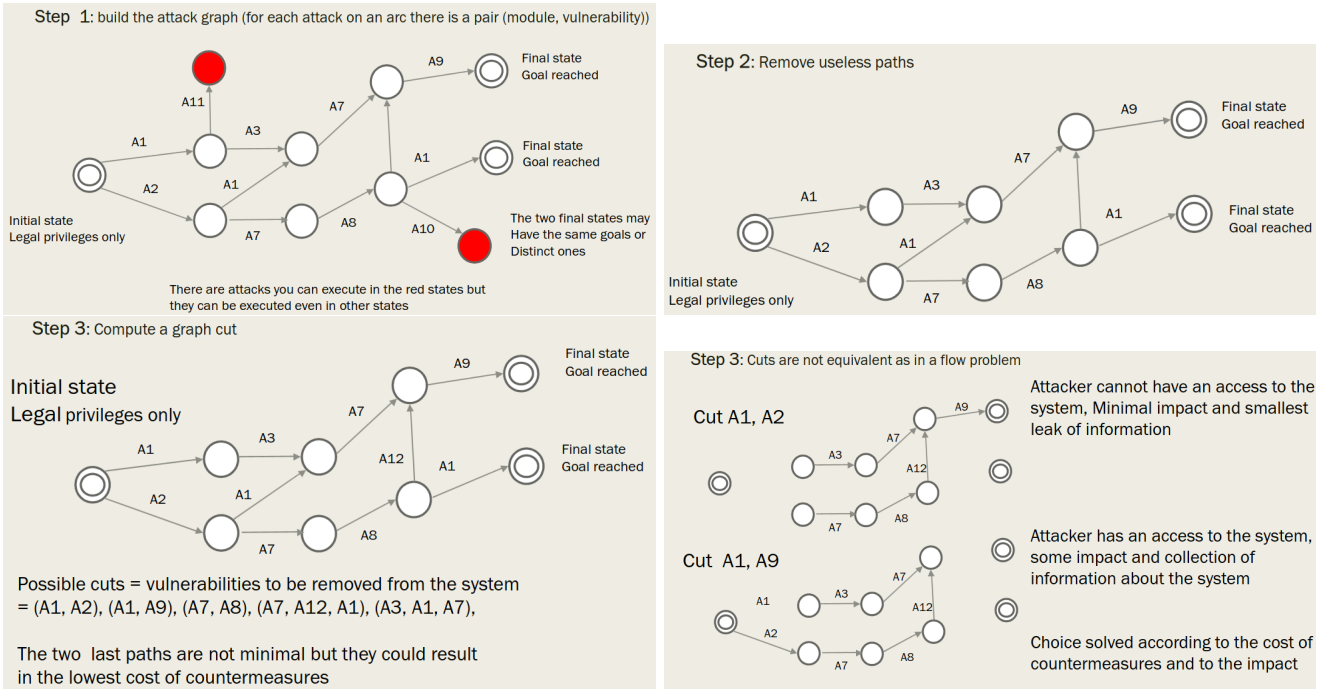


Figure 16.3: Stopping an intrusion using graphs

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.