ICT RISK ASSESSMENT

Vulnerability: a defect in a person, component
used by Threat Agent *(source of attacks to gain control of ICT resources)*

Unconditional Security *(cost and complexity aren't relevant)* **VS** **Conditional Security** *(security assessment)*

- **Risk Analysis**
  1. **Asset Analysis** *which resources to protect (CIA)*
  2. **Vulnerability Analysis**
  3. **Attack Analysis** *collect info on system -> discover vulnerabilities -> exploit -> remove traces*
  4. **Threat Analysis**
  5. **Impact Analysis** *Combining 1+2+3+4+5 == Risk Assessment*
  6. **Risk Management** *Classify risk, define acceptable ones, implement countermeasures.*

Security ≠ Safety *(randomness)* *1000 safe states, 1 unsafe = it's safe but not secure*

Example: Buffer Overflow

Low level C code with memory management and then writing larger data than expected.

Heap ↓

Text	*read only, stores the code*
Data	*stores static and dynamic variables*
Stack	*stores data to manage fun calls and returns*

Overwriting the return address with a valid memory address

Countermeasures? Strong typing, check strlens, "canary" (random chosen value before any parameter) into the stack , not-executable memory, ASLR, etc. Each one has a different cost

Example: DDOS

When defining TCP/IP, the main goal was resilience against attacks (real ones, like bombs) == AVAILABILITY

So when sending ECHO msg to check another node, receiver will reply with the same msg. It's possible to specify a partial IP to broadcast it ("Who is alive (from this partial IP address)?"). But what if A sends ECHO to partial IP X and puts B as the packet sender address? 😊

Countermeasures? Meh, this is a structural vulnerability, not much can be done.

So **System Design** is very important. It can be done in two main ways:

1. **Penetrate & Patch**: *pretend that there are no vulnerabilities in the components as an axiom. It kind of starts a competition between the attackers and defenders, who finds the vulnerability first and exploits it or patches it. In this case, each vulnerability is critical.*
2. **Proactive Approach**: *be aware of vulnerabilities and try to anticipate them. In this case, only unexpected vulnerabilities are critical if and only if there's a non-0 risk.*

1. Asset Analysis

Discover the assets that result in an impact if successfully attacked. This means approximating a value and of course isn't easy *(e.g. the cost of rebuiling it from scratch)*.

Prototypical cases	**Total Effort**	**Weakest Link**	**Best Shot**
Security depends on..	*Sum of individual efforts*	*Minimum effort*	*Maximum effort*

In any case, we need to define **Security Policies**: *a set of rules to minimize risk.* It's a critical point, it defines goals and assets of an organization, legal behavior for each class of users, roles, responsibilities, etc.
So we need to define **Subjects** and **Objects**. Subjects can invoke objects, and objects can invoke other objects too. **Subjects have rights** which are either **direct** *("S reads F")* or **indirect** *("since S reads F, any program run on S can read F").*

A standard start for Security policy is the use of **Default Allow** *(very dangerous!)* or **Default Deny** *(forbids anything that isn't defined).*
About access control, we defined the **Discretionary Access Control:** each object has an owner. This has then evolved in **Mandatory Access Control**, where objects and subjects are partitioned into classes *(maintaining a partial order)*. Some famous examples are:

- **Biba Integrity Model (No Write Up, No Read Down):** *Minimize info at high level*
- **No Write Down:** C can read any file with a class lower than C, can write = C and append > C. *Prevents loss/leaks of info, but the amount of info increases a lot.*
- **Bell-LaPadula (No Write Down + No Read Up):** write ≤ C, read > C. *Integrity over confidence.*

2. Vulnerability Analysis

A bug or violating any security policy are considered a vulnerability. To find them, we need to scan our system *(composed by standard [we know these] and specialized components [study these!])*

Vulnerability scanning *(fingerprinting)*

Test the *accuracy* = *(#TT + #TF) / (#TT + #FF + #TF + #FT)*

We don't always have the source code, so we test inputs reactions by either:

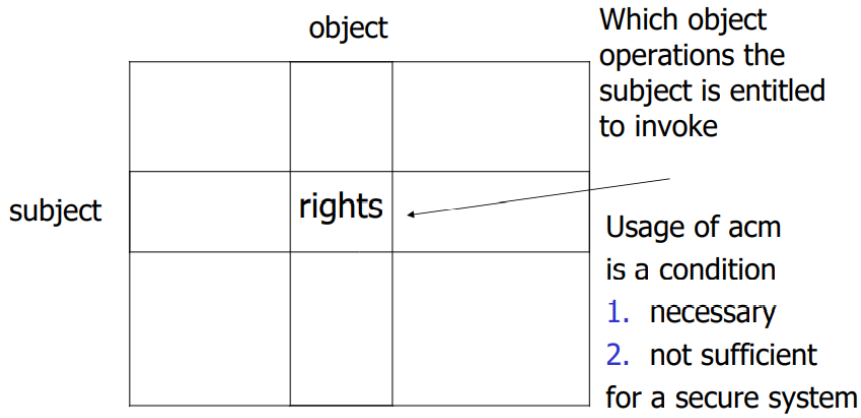
- **Tainting analysis:** computes a set of program variables that receive an input. *A larger set than the actual one implies FALSE POSITIVES*
- **Fuzzing:** applicable even if no source code. Send malformed data and look for crashes, etc. *(SPIKE, Sulley, Mu-4000, Codenomicon, Peach Fuzzer, ..) It can be a variant about mutation, generation, evolutionary, etc.*

After this we define the **Robustness of a module [0,1)**, which is its ability to avoid damage when specifications are violated. This can be increased only by decreasing efficiency!

8 (+2) principles

1. **Economy of principles:** KISS rule, complexity can be achieved by composition. *Esokernel, microkernel to avoid cascade failures*
2. **Fail Safe Default (default deny):** induction principle for security anytime the initial state is safe. *If the protection system fails, legitimate access are denied but also illegitimate ones.*

3. **Access Control Matrix:**



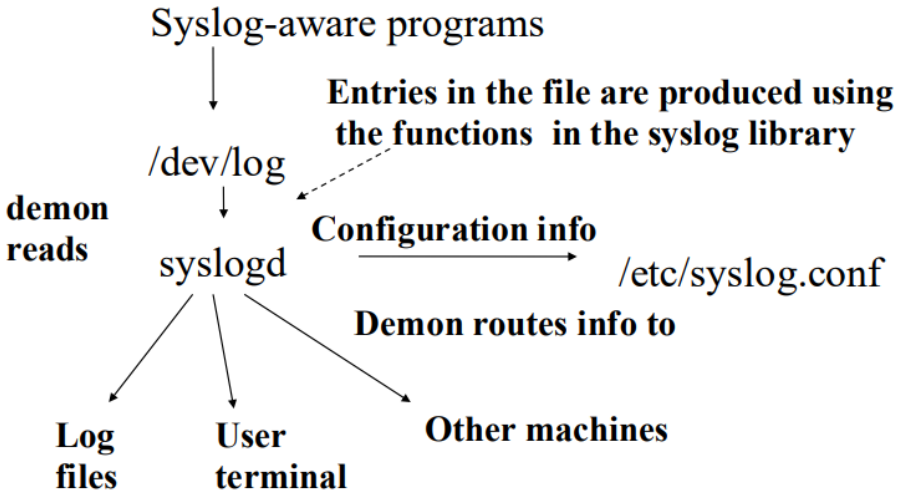
The main idea is to be fail safe by default, so that the system may start in a secure state. Then we need to check that someone who claims to be A is actually A: *this can be achieved by explicit checks in security kernels, passwords, one-time-passwords, challenge-response, electro-signatures, etc.* But in the ACM case, if the number of subjects increases, it's complexity increases too fast. So we need to define **Roles** to define **Role Based Access Control** (rights are assigned to roles, not users. Roles may be partially ordered) and **Attributes** to define **Attribute Based Access Control** (each subject is paired with a set of attributes).

But it may happen that we don't know all subjects beforehand, so we can use a Trusted Computing Base (TCB) in a tamper proof-memory *(which is a component where physical attacks are prevented or at least detected. Some examples are components glued with silicon, an electrified grid, etc).*

4. **Open Design:** the security shouldnt' depend on the secrecy of the design/implementation. This **isn't equal to** "source code should be public" *(need peer review and any peer communicates vulns)*.
5. **Separation of Privilege:** Complex operations are decomposed in simple operations. Check that all rights are owned. It requires the **Separation of Duty** and the **Defense in-depth**
   - *1 hub* flat network: any node interact with other
   - *n hub* segmented network: attack one by pivoting: *requires larger #atks (so more work)*
6. **Least-Privilege:** owning a useless right is a vulnerability, so rights are granted as needed and then **revoked**. *It defines how rights should be managed rather than how they're assigned* (UNIX violates this, root has all rights!). This way, even if an object is attacked, it'll have low impact!
7. **Least Common Mechanism:** minimize the amount of mechanisms common to more than one user and depended on by all users. *Isolation by virtual macchines / host and network segmentation.*
8. **Psychological Acceptability:** don't adopt policies users might/will violate. Furthermore, security mechanisms shouldn't increase the complexity of accessing a resource.

Last two principles introduced because even if the first 8 are satisfies a vulnerability may arise

9. **Work Factor:** compare the cost of circumventing the mechanism with the resources of a potential attacker. *The more resources an attacker can access, the higher the probability of a successful attack. Most attacks require a privilege escalation.*
10. **Compromise Recording:** mechanisms that reliably record a compromise of information may replace more elaborate ones that completely prevent loss. In any case, **Logging** *(append-only file)* is the key part. *Hash, blockchain, print it, whatever.*
    What happens when the log file is full? *Throw away, reset (from start), rotate (among files), compress, ..*
    Syslog: a logging system to store information produced by the kernel and by system utilities.



The demon implements the logging and is programmed through the conf file

Selector <TAB> action

All predefined in sets:

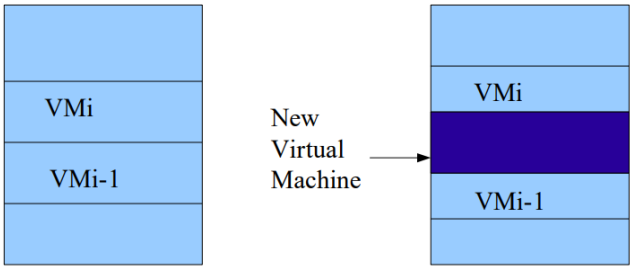
*The selector identifies the **source** and the **message severity level** (kern, user, mail, daemon, auth, ..), facility.level instead (emerg, alert, crit, err, warning, notice, info, debug)*

Anyway, all the principles don't fully characterize ICT security, because its two peculiar features are automatic attacks and the virtual machine hierarchy.

So first of all, any ICT system is a hierarchy of VMs, where each VM defines a set of mechanisms that abstracts and hides those of the underlying machine. Of course, we could go "standard" and consider known vulns, but remember that any vuln at level X implies attacks against any machine on top of X.

So a common attack is trying to attack at low level, maybe even inserting a new (malicious) VM!

Example: Blue Pill Attack



NVM can return fake infos about the system states to upper layers, transmit commands that differ from received by higher VMs, …

This implies the need of **robustness** at any level: each VM should include checks on the subjects/objects of its corresponding level, while still minimizing overhead. This may mean redundancy too, when checks are repeated in distinct VMs.

Shared memory areas among several applications by distinct applications *(which may not know in advance to be in a shared environment)* are indeed a problem, just by accessing an area and reading values left in it by someone else *(see cloud security)*. The solution is that when releasing any memory area it should be reinitialized to avoid any information flow.

3. Threat Analysis

Determine the enemies of a system. If there are no threats that can exploit a given vulnerability, then the assessment may neglect such vulnerability.

For each agent, it must determine its goal *(rights on components)*, the resources it has available, risk attitude, etc . Agents may be partially ordered *(resource they have access to, risk-willing, ..)*, so attacks aswell.

Each agent is described by a tuple of attributes and a noise. This way we define ***feasible attacks*** *as:*

Given

- a tuple T_A that describes the attack A and where each element evaluates an attribute of A
- a tuple T_M that describes a threat agent M and where each element evaluates the resources that M can access

M can execute A provided

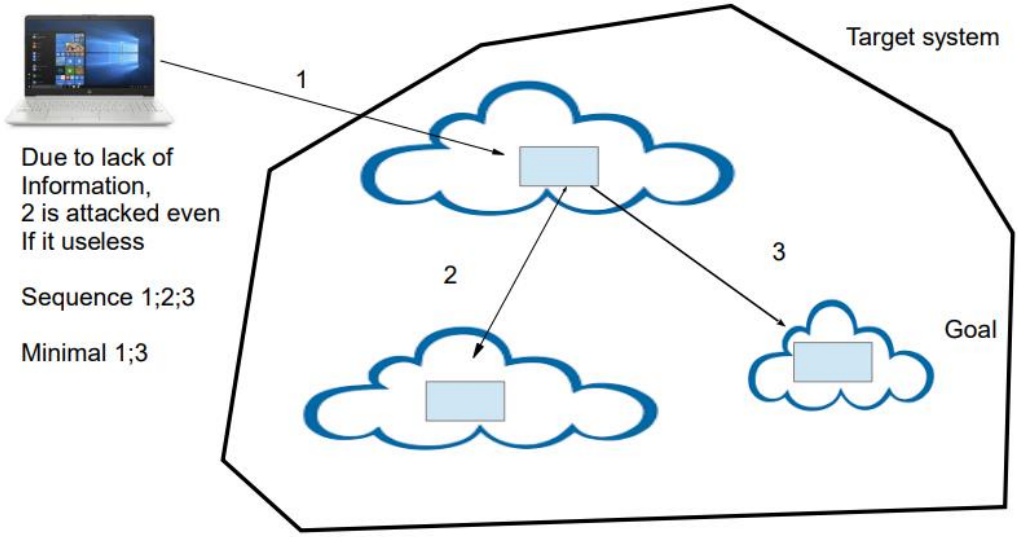
- Each element of tuple T_M is larger than or equal to the corresponding elemen of T_A
- The noise paired with A is smaller than or equal to the one that is accepted by M

The problem is to formally define the actions that any agent can/can't/won't execute. Considering that in ICT attacks are automated and in mass *(autonomous)* and usually the attack itself is the reproduction of the malicious agent, we should consider also how the attack occurs.

https://www.zerozone.it/tecnologia-privacy-e-sicurezza/google-dorks-il-potente-lato-oscuro-del-motore-di-ricerca/16843
lmao: https://www.exploit-db.com/google-hacking-database

4. ATTACKS (eh oh l'ordine delle slides è questo)

Attacks may be elementary or complex. Goals are in any case achieved by acquiring distinct access rights on system modules, resulting in an attack plan. A plan is **minimal** if all the attacks it includes are required to reach the goal. An example of a "non-minimal" plan is shown in figure, due to lack of information.



Any attack can be modelled through *(at least)* 6 attributes:

1. **Precondition:** *rights on system modules*
2. **Postcondition:** *still rights, but after*
3. **Enabling vulnerabilties**
4. **Actions to be executed**
5. **Success probability**
6. **Noise:** *the detection probability, events the atk generates that make the atk detectable*

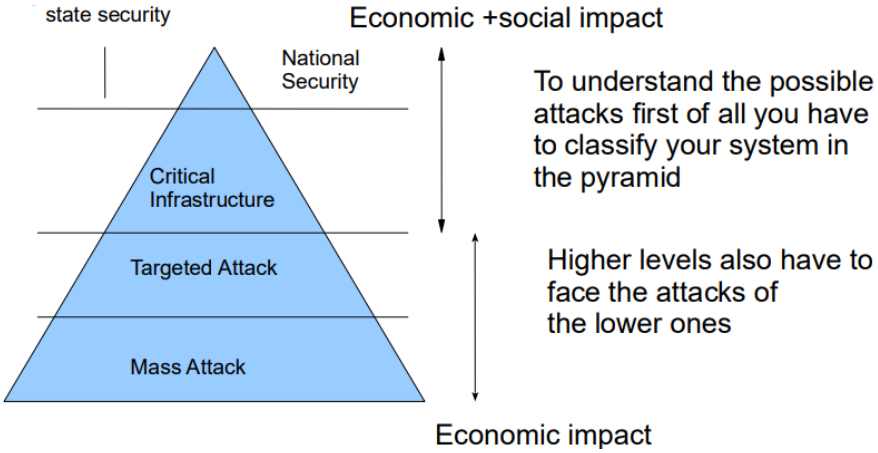
In an elementary attack, the postconditions are the set of rights granted later (including the pre-ones).

Example: stack overflow
Here you need to be able to invoke a procedure, to know the memory mappings, etc. The noise here is the potential crash when attacking the stack "wrong".

More elaborated attacks are related to cryptography. A taxonomy of these: *Brute force atk, Differential cryptanalysis, Linear cryptanalysis, Meet-in-the-middle atk, Chosen-ciphertext atk, Chosen-plaintext atk, Ciphertext-only atk, Known-plaintext atk, Power analysis, Timing atk, Man-in-the-middle atk.*

Open framework for Common Vulnerability Scoring System (CVSS): https://www.first.org/cvss/

Anyway, it's meaningless to evaluate attacks independently of the target system, so lets classify them.



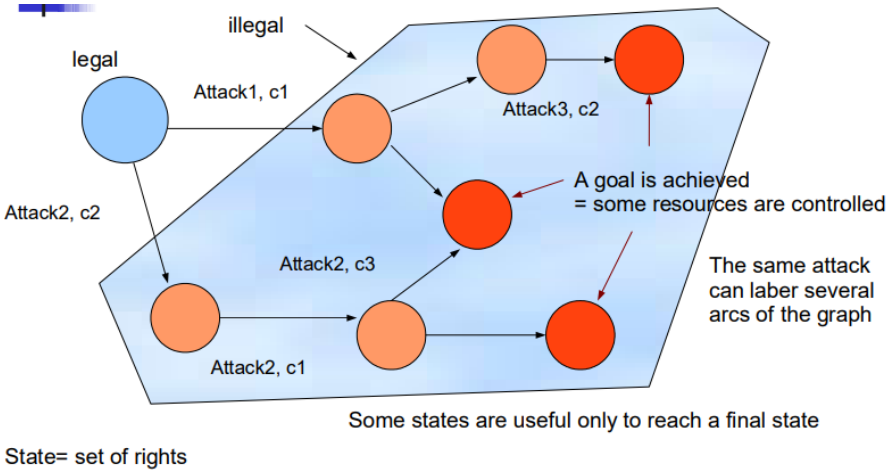
Mass attacks: take advantage of the openness of Internet *(phishing, water holing, randomware, scanning)*

Targeted atk: tailored to attack systems/processes *(spear-phishing, DDOS deliver, ..)*

Note that in a complex system a threat cannot achieve one of its goals through just one attack. This means that it exists an attack chain, where preconditions for each sequent atk are enabled in the "last" attack. This usually means that these actions are needed:

- **Host discovery:** *who are the hosts in a network*
- **Topology discovery:** *msg routing in the network*
- **Vulnerability discovery:** *the vulnerabilty of an host*
- **Attack selection:** *choose the atk to execute*
- **Failure handling:** *handle an atk failure*
- **Defence evasion:** *avoid defence mechanisms*

This way the attacker can define a model that describes the *system components, system interconnection structure,* and finally *component vulnerabilities.* This will then generate an attack graph (*monotone because of rights acquisition and it must consider the worst case where attacks are successful*)



Of course this may cause state explosion. The number of paths/links/nodes of the graph can be strongly reduced by **focusing on an attacker behavior (Monte Carlo Analysis)**: *we choose only paths that we know attacker will use according to its preferences and priorities (non deterministic).* [instead of the graph, we could use the tree too, where every subtree is an attack *(note: they may collide)*]

Countermeasures are either static *(change the system forever)* or dynamic *(change the system only when under attack)*. Since we can stop a complex attack by stopping any of its elementary ones, we define the cut sets of an attack graph, which is a set of arcs *(elementary attacks)* such that no goal can be reached if they're cut. Of course many sets may exist with different cost: choose them depending on the most shared elementary attacks or the cheapest way of doing it.

Malwares:

Virus: a program that hides itself in another program/data. Two variants: **encrypted** *(constant decryptor followed by the encrypted virus body)* or **polymorphic** *(each copy creates a new random encryption of the same virus body, but the decryptor code is constant and can so be detected)*. An antivirus would look for signatures (fragments of known virus code), it would look at heuristics for recognizing code associated with viruses *(e.g. polymorphic uses decryption loops)*, or it would check integrity to detect file modifications. Another techinque is the CPU emulation for a few hundred instructions and recognize known virus body after it has been decripted.

The next step of course is to mutate the virus body as well, **metamorphic** virus. It carries its source code *(useless)*, it looks for the compiler on the infected machine, then changes its source and recompiles itself.

Anyway, every malware has some sort of code obfuscation to prevent code analysis, signature-based detection and foil reverse-engineering. *Different code in each copy of the virus, due to the "undecidable problem" it makes the analysis difficult by passive/static tools.*
Some of the most used: instruction reorder, branch condition reversed, different register names, NOPs and jumps in random places, garbage opcode in unreachable code areas, sequences functionally equivalent.

Example: Z-mist
Through code integration, it would merge itself in the instruction flow of its host. It would integrate in random locations its code and link it by jumps: then when the virus code is run, it would infect every

available portable executable. To integrate itself, the virus must disassemble and rebuild host binary *(tricky due to offsets)*.

Worms: automated and autonomous attackers that can replicate onto attacked nodes (through remote attacks). The code to infect other nodes is the *attack vector*. Due to all the noise they create while infecting *(even if slow atk is implemented)*, it has to consider the two disjoint subsets for address generation: local (high density, subnet of the infected node) and global (low density). The *density* is the probability that a random address belonging to the set corresponds to a real node. If the ratio of local vs global addresses is too low, the worm may be detected and removed before spreading. If the local density is too large, then after infecting all nodes resources are wasted because one node may be infected several times *(non linear)*. The model is epidemiological *(see SIR model)*.

Trojan horse: its main goal is to implement a backdoor to enable illegal accesses to the system. This leads to a dog chasing its own tail, because even with some defense (SODDI, *some other dude did it*), anyone looking at source code would see it -> change the compiler to add backdoor at compile-time -> but looking at compiler source code… -> change the compiler to recognize when it's compiling a new compiler and to insert trojan into it -> … . **So, you can't trust code you didn't totally create yourself**. Famouse TH are **rootkits**, their main characteristic is stealthiness because they install hacked binaries through running standard UNIX commands. Then they may re-route a legitimate system function to the address of malicious code *(pointer/detour/inline hooking)*.

5. Impact Analysis

It determines the loss for each successful attack. *Some loss can't be quantified or are hard to do it.*

Proper questions to ask: *how long could the business continue to operate without that particular system? What would be the opportunity cost of that downtime? What would be the real costs to bring it back?*

The **impact** is modeled as a random variable with a probability distribution and it's the average of the distribution. A very popular distribution is the *normal* one.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

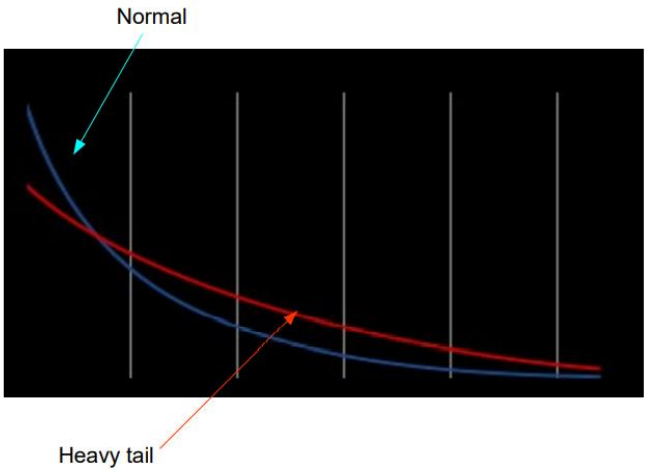
mu = mean of x, sigma = std dev of x

This way there are several nice mathematical properties (*e.g. central limit theorem)*. Furthermore, it represents also a thin tail *(exponential decrease in the value of the impact)* which is considered when values are 3-4 stddevs away from the average value *(in other words, it's centered around the average, which may be too optimistic)*.

Mild Randomness (Mandelbrot): models a process that is the sum of several other processes and where each process may compensate the other ones.

Mediocristan (Taleb):

Heavy tails: when the threat can achieve any impact if in full control of the system and the owner can't regain control. In these cases, the impact can't be modeled by a normal distr. So Power-Law *(cascade failure is where the heavy tails await)*



Example: the impact of an attack that results in the knowledge of some credit card numbers can be modelled through a normal low. The impact of an attack against a server that stores all the credit card numbers can't be modelled nor bounded.

Note: *the difference between Power-Law and Normal can't be discovered in an experimental way because if the number of data available is very low this may result in a too small difference. So decide according to the features of the attack.*

Solution? Redundancy. It increases the complexity of attacks *(redundancy in controls)* and decreases the impact *(redundant resources)*. *But it's effective IF the independence against attacks is guaranteed!*

Cryptography (Symmetric & Asymmetric)

The security should rely on algorithmic properties (math), not obscurity. *No secrecy of implementation, but rather secrecy of a key.*

Stream ciphers *(one byte of plaintext at a time)* are way faster than block ciphers. It approximates the one-time pad with the practical part of finite keys through a keystream *(infinite sequence of random bits)*. *The most popular one is the RC4, 10 times faster than DES.*

RC4 implementation

Table initialization

- for i = 0 to 255
  S[i] = i
  j = 0
  for i = 0 to 255
  j = (j + S[i] + key[i mod kl]) mod 256
  swap (S[i], S[j])
- kl=keylength

Encrypt/decrypt

- i = 0
  j = 0
  for l = 0 to len(input)
  i = (i + 1) mod 256
  j = (j + S[i]) mod 256
  swap (S[i], S[j])
  output[l] =
  S[(S[i] + S[j]) mod 256]
  XOR input[l]

Client & server should use different keys, because knowing one you can figure out the other.

About asymmetric we introduced RSA, elliptic curve cryptography *(elliptic curve discrete log problem)*.

Finally, we spoke about key management & exchange (PKI, and types of keys: *identity (AUTH), session (C), integrity (to compure MACs and used in digital signatures*))
Reminder: *MACs used to determine sender of a msg. If A & B share k, then A sends msg M with MAC tag t = MAC(M,k). Then B receives M' and t' and can check if signature is tampered by verifying t' = MAC(M',k)*

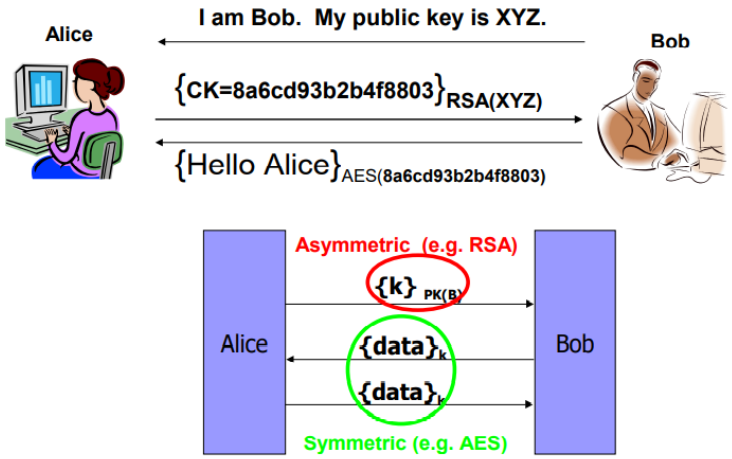
To securely generate keys, in WindowsOS *CryptGenKey()*, in Java *SecureRandom*.
But where to store them? Surely not in source code *(reverse-engineering)*, not in a file *(can be read)*, not in a registry *(can still be read, regedit)*. They should be stored in devices external to computer, so that the key won't be compromised even if the computer is!

- **Smart card:** limited CPU, vulnerable to power atks, rely on using untrusted PIN readers.
- **Hardware Security Module (HSM):** key never leaves it, higher CPU, dedicated
- **PDA or Cell Phone:** no intermediates like PIN readers, more memory, fast, can have its own bugs.
- **Key Disk:** no CPU, not tamper-resistant, no support for auth *(USB, non-volatile mem, 2nd auth fact)*

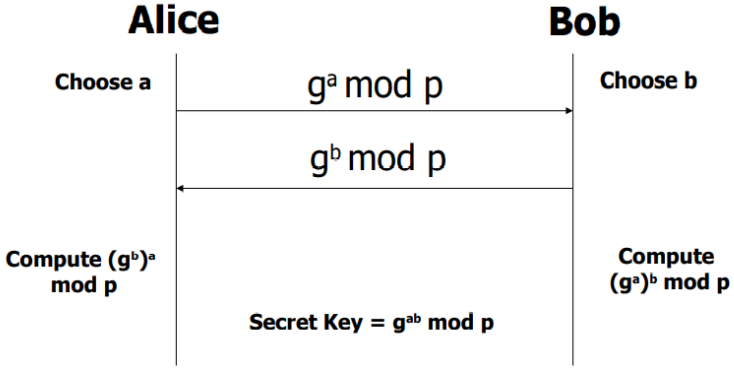
So ideally if the crypto operation is done on the device and the key never leaves it, the damage is limited, because otherwise its connected to a compromised host and ¯_(ツ)_/¯.

To **share** these securely generated and stored keys, we then must create a secure communication channel.

1. **Use asymmetric keys:** public key cost >> symmetric one. So use RSA to send cryptographically the random conversation key k, then use k as a key for faster symmetric ciphers *(e.g. AES)*.

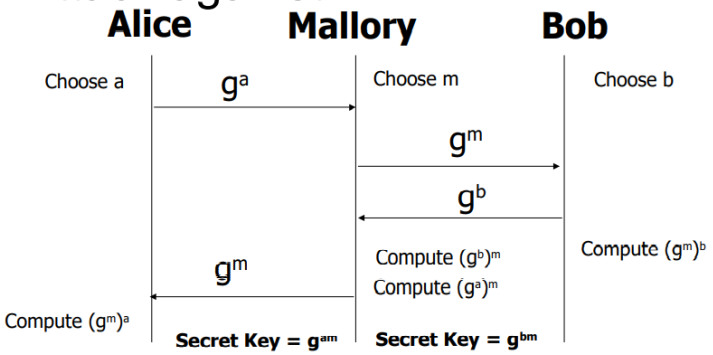


2. **Diffie-Hellman key exchange:** use public parameters *p (large prime number)* and *g (generator of Z_p = {1, …, p-1})*. Then Alice & Bob generate random #s a, b respectively and then using *g,p,a,b* they can create a secret known only to them.



Eve can compute $(g^a)(g^b) = g^{a+b} \bmod p$ but that's not the secret key!

14.4.2. Man-in-the-Middle Attack against DH



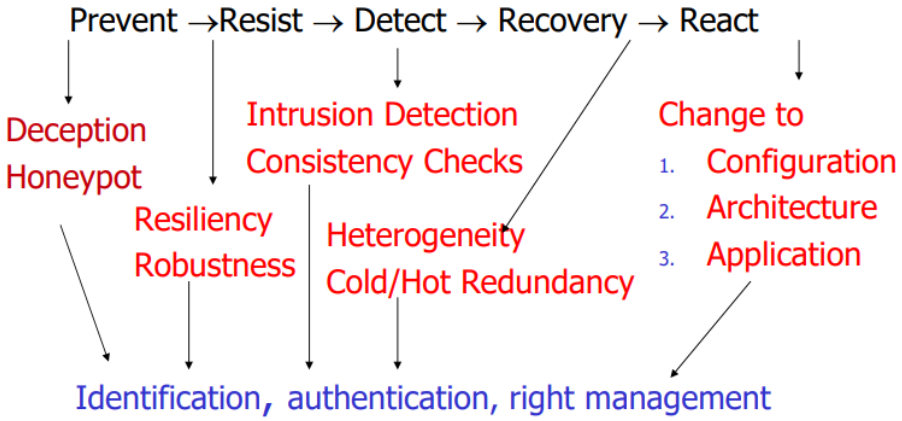
Mallory can see all communication between Alice & Bob!

Finally, we spoke about other stuff and SSL handshake *(Client & server agree on master secret used for symmetric crypto through 2 round trips. The first is "hello" msg -what version of SSL and which crypto algo are supported- then the 2nd varies based on client / mutual auth)*.
Note that cryptography doesn't solve a problem: you can't hide a 1Tb file, so you encrypt it with a 512 bit key. Now you gotta hide the key! *Same problem, but simpler.*

300 slides di **COUNTERMEASURES** in arrivo aaaaaaa /|/'''(•●)

How to change the target system to avoid or at least minimize the risk.
Proactive (before an atk), **Dynamic** (as soon as atk is detected), **Reactive** (if atk is successful)

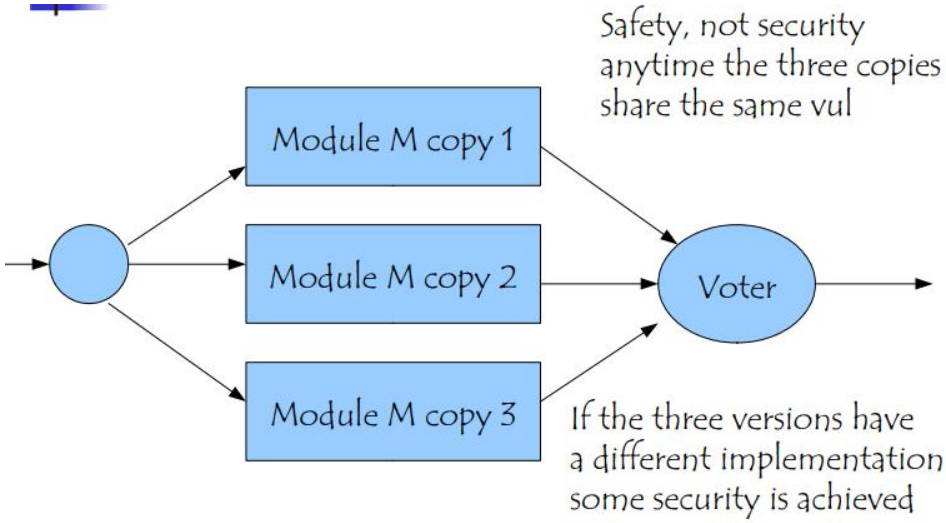
Detailed taxonomy:



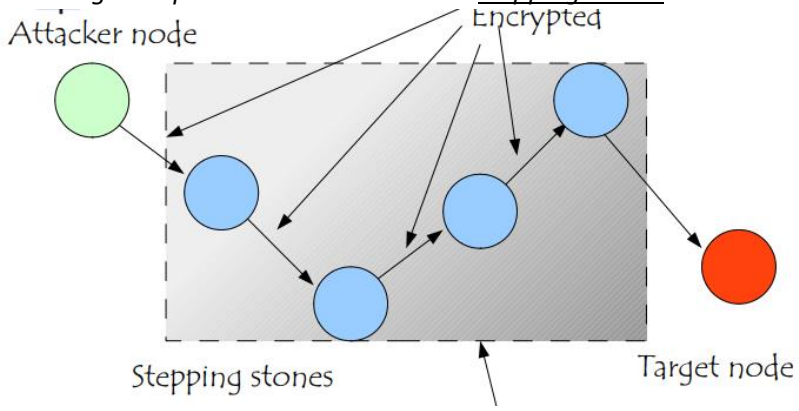
Countermeasures are implemented through a set of **common mechanisms** *(which can increase the cost effectiveness of countermeasures because a vulnerability in one may affect several countermeasures -> robustness needed)*. These mechanisms are defined on top of a security kernel (TCB) that manages the user identities, authentications and rights.
Glossary:

- **Deception:** *no information about the system design is available*
- **Honeypot:** *fake systems to increase the complexity of discovering nodes to be attacked*
- **Resiliency/Robustness:** *prevent a single vuln from enabling a successful attack*
- **Intrusion Detection/Consistency check:** *check to discover current/previous attacks*
- **Redundancy:** *spare components to replace attacked ones. Cold stand, hot in use*

- **Heterogeneous components:** *genetic diversity. Different components have different vulns*
- **Triple Modular Redundancy:**



- **Minimal System:** *subset of components, more robust, large number of severe checks to prevent losing control of the system. It's a starting point to gain back control of it!*
- **Reaction:** *updates to system architecture, application, patches, configurations, etc. It usually doesn't involve the attacking one*
- **Offensive security:** *react by attacking the attacking system. Eeeeeeh. Who made the attack? A chain of hosts (and the attacker is at the start): the chain hides him, you can only see the last node that maybe is just being manipulated! These are called stepping stones.*



To find these stepping stones, do an analysis of **input/output** node channel to evaluate their correlation. If time *(when communication occurs)* and data *(size of exchanged data)* are correlated, then the node may be a stepping stone. Repeat the analysis to discover the whole chain.
Otherwise, use an **honeypot**: they are useless virtual nodes, but for the scanning that the attacker has to analyze, they behave like real ones. *They'll reply to the fingerprinting msg with frequency that is slower and slower to slow down the scanning and even raise alarms.* Definition (Honeypot): an ict resource whose value lies in unauthorized or illicit use of that resource. Only for monitoring/detecting/analyzing!
Honeypots **classification**:

a. **Level of Interaction**
    i. **Low:** *simulates some aspect of the system, has limited informations, it's easy to deploy, minimal risk. Example: a simple port listener. Once the attacker connects, he can't do anything else*
    ii. **High:** *simulates all aspects, can be completely compromised (higher risk), has more infos*

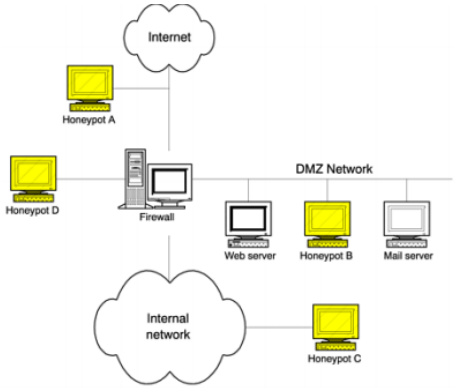
iii. **Middle:** *example, an emulated service that analyzes communications and returns simulated responses to replicate real services*

b. **Implementation**
   i. **Virtual:** *simulated by other machines that respond to the traffic sent to the honeypots and may simulate distinct virttual honeypots at the same time*
   ii. **Physical:** *real machines, own IP addresses and often highly-interactive*

c. **Purpose**
   i. **Production:** *prevention (keep atker out, doesn't work against untargeted attacks like worms, auto-rooters, etc), detection (it means detecting the burglar when he breaks in #greatwork), response (can be easily pulled offline and gives little/no data pollution).*
   ii. **Research:** *develop analysis, discover new worms/viruses/signatures, collect compact amounts of high value informations, etc.*

How to build an honeypot: specify goals, implement data capture *(host based: keystrokes, syslog. Network-based: firewall, sniffer)*, log, mitigate risk and fingerprints. *Anticipation - the firewall: a system that connects two networks with distinct security requirements, by filtering informations flowing across them. It hides some components so that they cannot be accessed from the less critical network, while defending the most critical one from attacks. Honeypots are then put:*

- In front of the firewall
- Demilitarized Zone
- Behind the firewall (Intranet)



Honeyd: a virtual honeypot application to create thousands of IP addresses with virtual machines and corresponding network services. http://www.honeyd.org/

So, honeypots use known vulnerabilities to lure attacks. **Honeynets** are networks open to attacks, in the hope that the attacker will mess up the honeynet instead of the production system (it's behind a firewall, uses default installations of system software, ..). It's a highly controlled net where every packet entering or leaving is monitored/captured/analyzed: it's suspect by nature.

- **Robust (Proactive) Programming:** validate program inputs *(Input validation + default deny)*, prevent buffer overflows, invoke only safe functions *(robustness)*, check invocations *(correctness of transmitted parameters and their metadata)*, check returned results *(wrong login? Don't say "incorrect password" but "<user,pwd> doesn't exist")*, safe variable inizialization, atomic transactions on file systems, mutual exclusion to preserve consistency, … . Most of these constraints can be enforced by the program run time support (Java) or be satisfied because there's a good programmer (C): both are ok, depending on privileging performance or security.

So, to Resist, the first step is to have a correct configuration of standard software component, determining useful functions and removing useless ones. Then introducing firewalls, which can protect a net from attacks from the outside, but **not** from the inside! The firewall is characterized by two features: robustness

enabled by the controls and robustness in the control implementation. *Note that the same set of controls can be implemented in totally different ways, either connecting or not two components.* The simplest firewall is a packet-filter *(both incoming and outgoing interfaces)*: you specify allowable packets in terms of logical expression + default deny and badabum you're done.

Example:
We want to allow inbound mail (SMTP, port 25) but only to our gateway machine. Furthermore, SPIGOT's site must be blocked.

action	ourhost	port	theirhost	port	comment
block	*	*	SPIGOT	*	*we don't trust these people*
allow	OUR-GW	25	*	*	*connection to our SMTP port*

Then suppose we want to implement "any inside host can send mail to the outside"

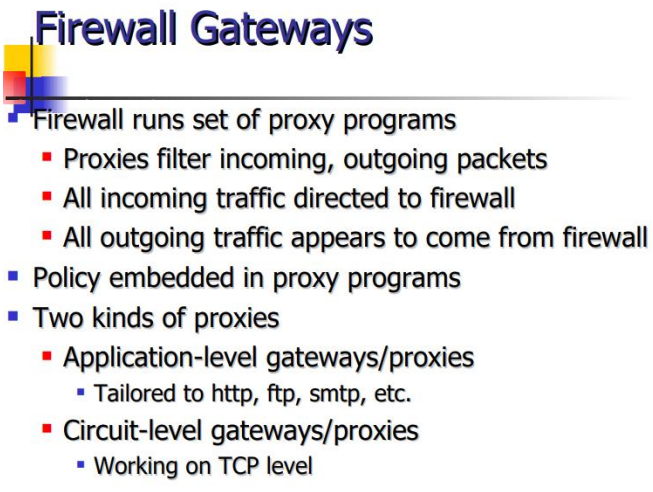
action	ourhost	port	theirhost	port	comment
allow	*	*	*	25	*connection to their SMTP port*

WRONG: the defined restriction is based solely on the outside host's port number, which we can't control. An enemy can access any internal machine and port by originating his call from port 25 on the outside machine. Solution: introduce ACK *(so the packet is part of an ongoing conversation)*. Packets without the ACK are connection establishment msgs, which we only permit from internal hosts!

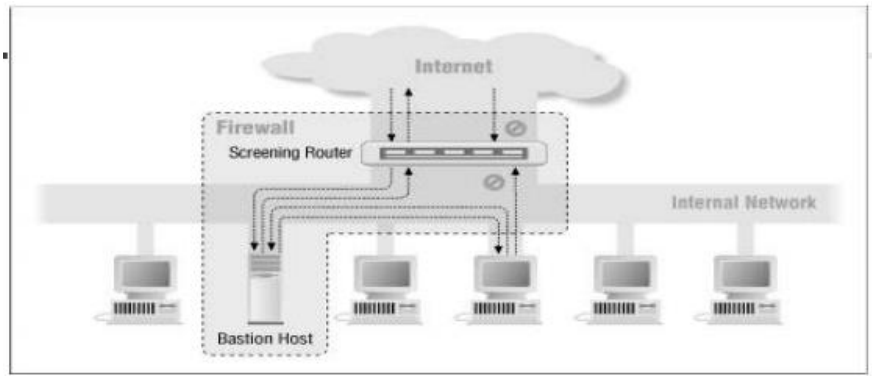
action	src	port	dest	port	flags	comment
allow	{our hosts}	*	*	25		*our packets to their SMTP port*
allow	*	25	*	*	ACK	*their replies*

An evolution of packet-filter firewalls are stateful packet filters: this is needed because traditional packet filters don't examine transport layer context. So these examine each IP packet in context, keeping track of client-server sessions.

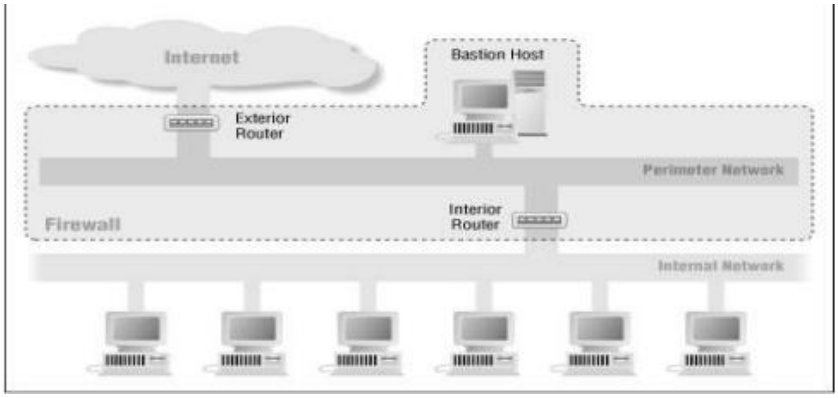
Then, application-level filtering: has full access to protocol, so it needs different proxies for each servive *(e.g. SMTP email, NNTP net news, DNS, NTP, ..).*



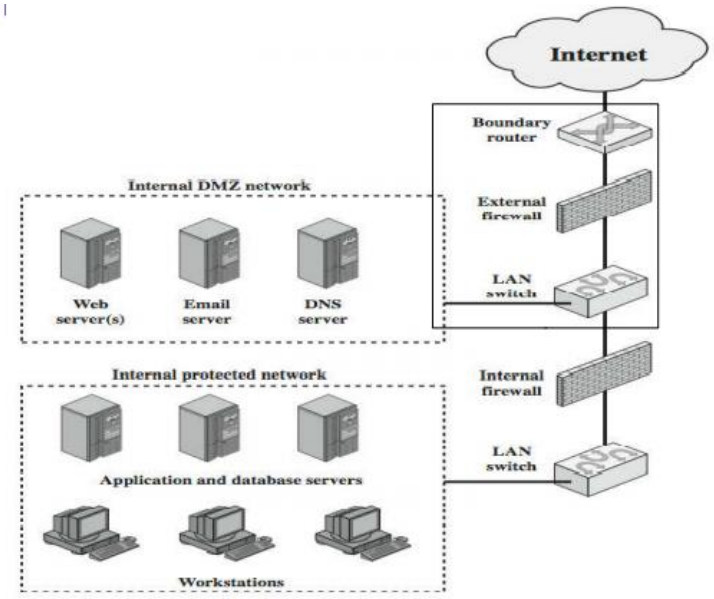
Another firewall architecture is the screening router + bastion host: the BH is the only system on the internal network the Internet can open connections to. Only certain types of connections are allowed. Of course, it needs to maintain a high level of host security



Otherwise the screened subnet is an extra layer of security by adding a perimeter network to further isolate the internal network from the Internet: only BH can be attacked.

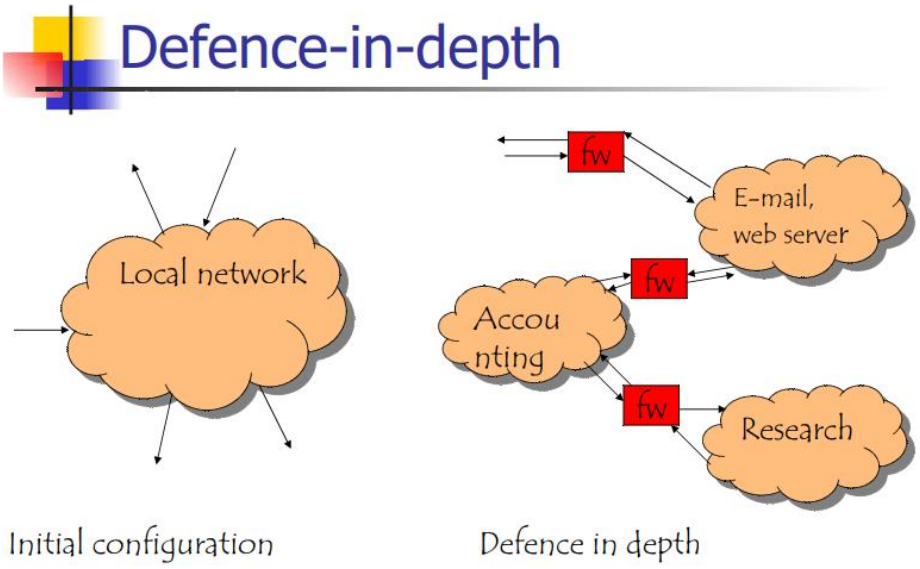


De Militarized Zone (DMZ): Layered protection == **defence in depth**

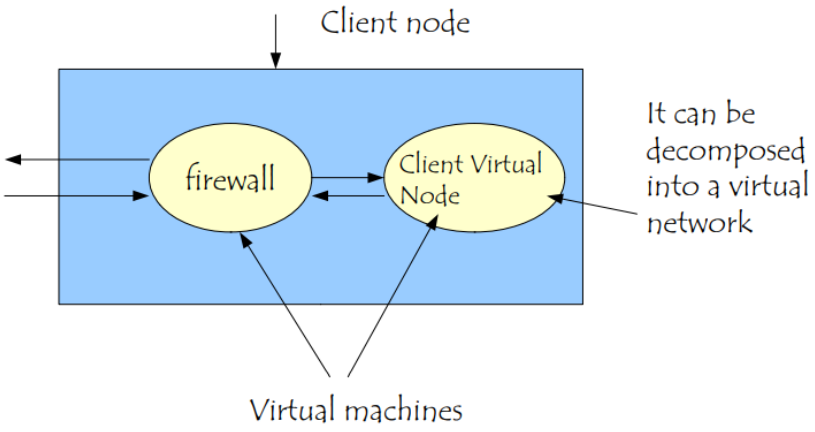


Even a simple router with packer filtering

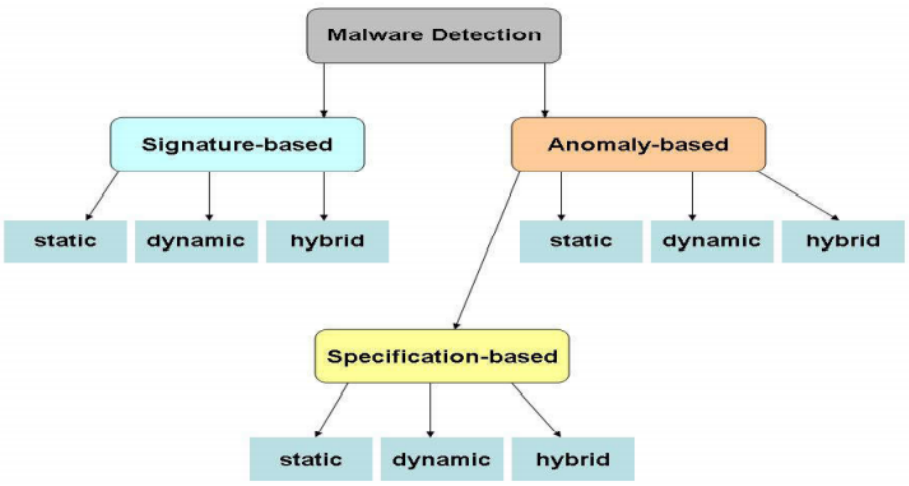
Pros: 3 layers of protection that segregate the protected network. To penetrate, must crack the outside firewall router, the bastion firewall and the inside firewall devices. The private network is invisible because there are no routes to it. Since it's different networks, a NAT can be installed. Resuming:



Anyway, things evolve from time to time. Initially, targets were server systems. Currently attacks are more complex, because they may want to attack a client system as an intermediate step to steal infos used to authenticate users. So another type of firewall, personal firewall: to protect the client and the information exchange between the client and the server.



These have two cases of interest: either discovering ongoing attacks or discovering malware that is already installed on a node. The screenshot below represents a general schema:



Anomaly based: observe the behavior of the system for an interval of time *(lean normal behavior)*. Then any behavior too "distant" from the observed is signaled as an anomaly. It can be Dynamic *(information on a program behavior is collected to discover attacks against it)*, static *(information on the structure of a program or of file record are collected)*, hybrid *(the expected behavior of the program is compared against the actual one)*.

Specification based: normal behaviors specified by the security policy. Again, Dynamic *(information on the program behavior are collected and compared against the program specification)*, static *(a program is statically analysed to compute the spec and the behavior is compared against the specification)* and Hybrid *(the compilation returns a specification integrated by observations to be compared agains the program behavior).*

Signature based: the main idea is that some behavior fully characterize and identify a malware (their signature). All of these are then collected in a DB: this poses two problems: the discovery of a signature and the update of the DB. There's a default-allow strategy *(anything that is different from a signature in the DB is allowed, but even default deny can be implemented)* and can be Dynamic *(information on the program behavior are collected and compared against the signature)*, Static *(the program code is analyzed and compared againt the signature, e.g. antivirus)*, Hybrid *(merge the two: a subset of programs is selected by a static analysis and the behavior of these is monitored)*.

Of course there will be false positives (and negatives)! -> ROC curve (AUC)
(specifity = prob of a false answer if no illness. Sensitivity = prob of positive answer if illness)

Host IDS: it monitors a single host, checking system and user process to discover changed OS commands, attacks, etc. The base mechanism is the interception of the OS call and either analyze it *(real-time)* or produce a log with the call and analyze that *(offline)*.

Network IDS: monitor the network segment inbetween two switches, to detect anomalous or dangerous traffic (sniffing, as an attacker does).

NIDS + HIDS: they can cooperate, but need mutual trust. It's important to determine whether two events are independent because if several independent events signal an intrusion, then the probability of a true positive increases. *To detect anomalies there are many interesting measures, like the number of open files, number of open ports (global, for each user), frequency of commands, number of connected users, time when a user connects, usage of system resources, … .* Then you put a user defined threshold (too large = few false positives, several false negatives. Too small = few false negatives, several false positives).

In any case, the use of an IDS has to be **transparent** for the user. But what can an IDS do? It can take action on the target system *(kill an internet connection, end user sections, etc)*, but can't do anything against other systems, even the attacker one *(both because stepping stone and false positives)*. *The NIDS isn't involved in the service that manages a given packet, so it can't slow down the receiving host, while the HIDS may slow down.*

Bypassing NIDS:

- **Fragmentation:** NIDS must reconstruct fragments == **maintain state and must overwrite correctly** (drain on resources!). Attack #1: just fragment. #2: frag with overwrite. #3: start an atk, follow with many false atks, finish the first one.
- **TCP un-sync:** inject a packet with a bad TCP checksum *(fake 'FIN' packet)*, or inject a packet with a weird TCP sequence number *(step up, wrapping numbers)*.

- **TTL attack:** it's an atk against the synchronization of the IDS: it requires a router between the IDS sensor and the end host. A packet crafted with a TTL equal to the number of hops of the router will result in a packet examined by the IDS but never reaching the end host, thus desynchronizing the end host and the IDS. The solution is that the NIDS should understand the network topology *(expense of a larger overhead)*.

Bypassing HIDS:

- **Stack protection:** a canary is placed next to return address
- **Overwhelming:** send as many false attacks as possible while still doing the real attack to overload console/hope to drop packets.
- **Slow Roll:** detect port scans and sweeps

Example #1: Snort, a good freeware network sniffer, useful for traffic analysis and intrustion detection. It uses a detection engine based on rules: if a packet doesn't match any rule, it's discarded, otherwise logged. The packet is checked in order: drop > pass > alert > log. This way every packet passes through the drop rules, but it's not very efficient. A more efficient way is Pass > Drop > Alert > Log, but it's more dangerous.

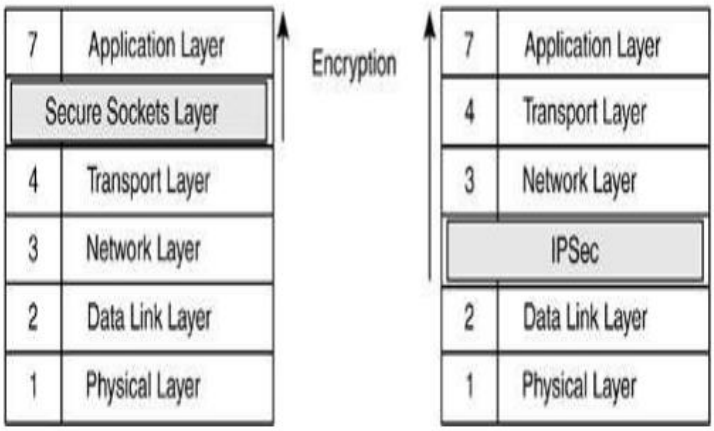
Example #2: Zeek, a network analyzer. It's rule-based too and it's used to record network behavior. Cons: it's deep-packet inspection == resource intensive.

Anyway, leased lines are too expensive and we still need to connect to remote networks. The most convenient connection exploits a public network *(e.g. the Internet)*, but the security here is very low because information flow on a public network. Countermeasures?

VPN: it emulates a secure connection on top of an unsafe connection. Assuming that each local network is protected by a firewall, secure connections are established among the firewalls. It's secure because C and I are achieved by (symmetric: e.g. the easiest one, Diffie-Hellman) encrypting the traffic between any pair of firewalls (so different keys for each pair). Cons: any implementation of any VPN may be the target of a DDOS attack. A VPN decrypts any msg it receives, so if receiving a flood of fake msgs the receiver will be busy discarding them and can't receive legal ones (no A).

Note: it's not a VLAN! This denotes a logical network that is setup to minimize the number of conflicts and has no security property.

IPSEC: An Ipv4 extension to authenticate and encrypt information flows (until Ipv6). It can be used between two hosts, two gateways or a gateway and a host. By replacing IP with IPSEC, we increase communication security in a more transparent way for the involved hosts, while requiring no updates to the SW/HW network components.

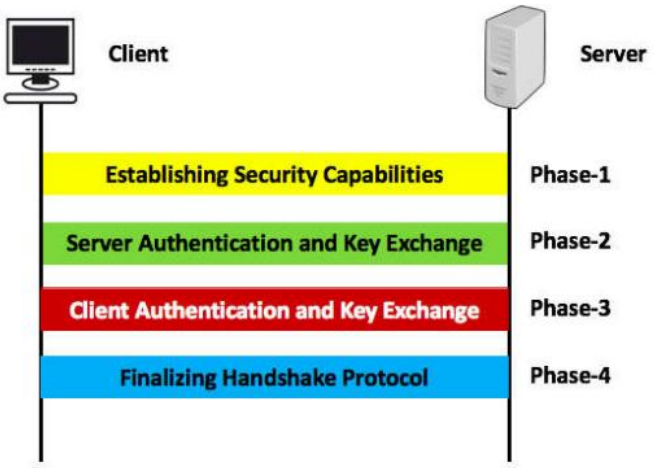


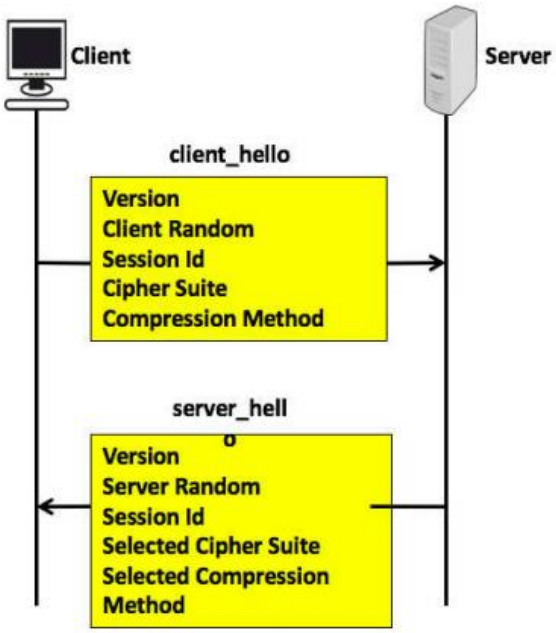
Glossary 2:

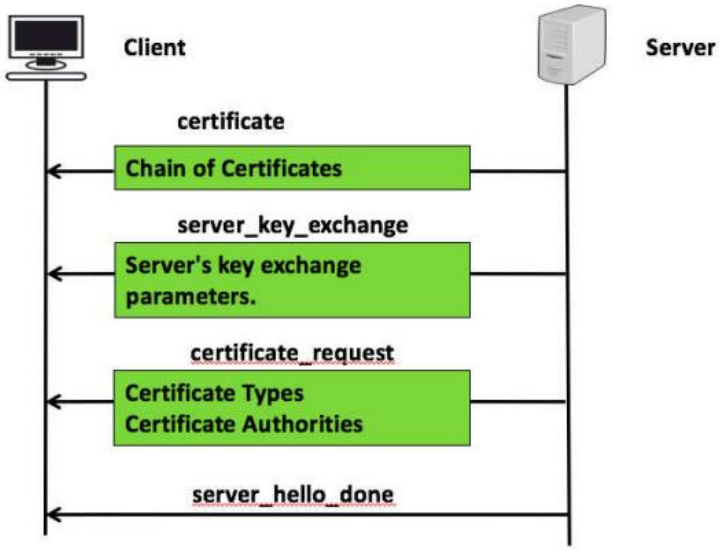
- **Session:** association between a client and a server that defines a set of parameters such as algorithms used, session numbers, etc. A session is created by the Handshake Protocol that allows parameters to be shared among the connections made between the Server and the Client. They allow to avoid negotiation of new parameters for each connection.
- **Connection:** logical client/server link. In SSL terms, it's a P2P connection with two network nodes.
- **Session identifier:** an arbitrary byte sequence, chosen by the server to identify the state of an active session and can be resued to continue the session.

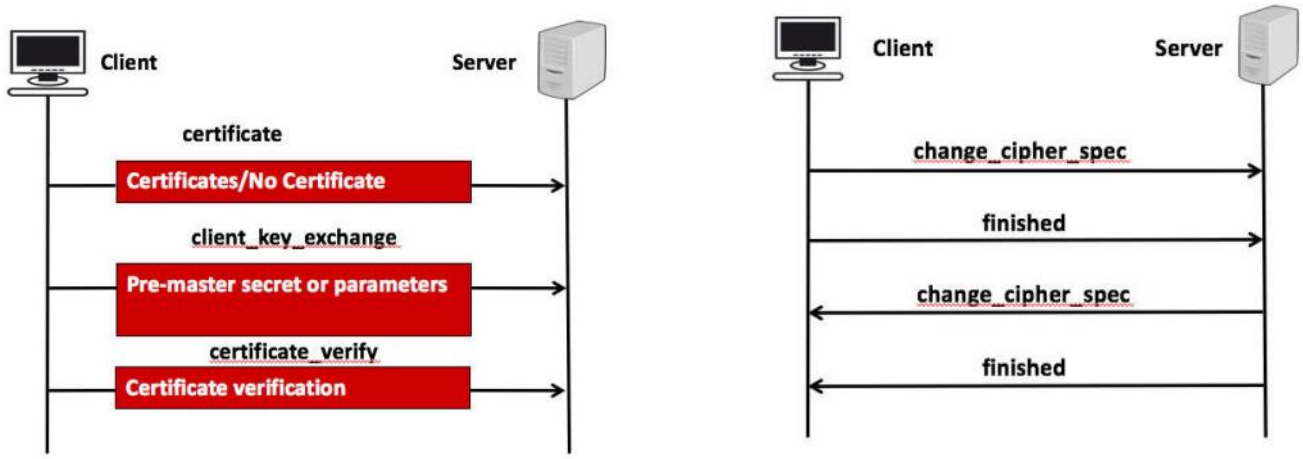
SSL

4 phases:



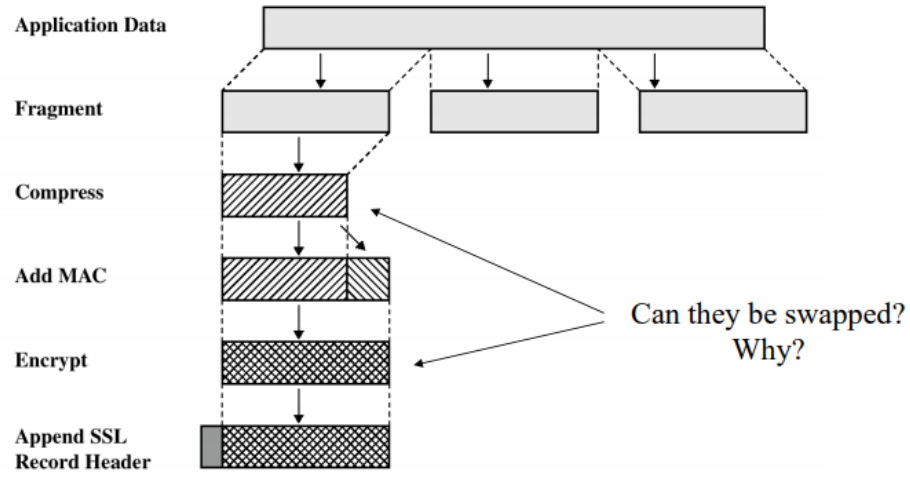






The 4th is the only session which enables multiple connections.

ChangeCipherSpec protocol: special protocol with only one message. When client processes encryption information, it sends a CCS msg (then followed by a Finished msg), signaling that all following msgs will be encrypted.



MAC secures connection in two ways: ensure client and Server are using same encryption and compression methods, and ensure that msgs sent were received without error or interference. Both sides compute MACs to match them and if there's no match -> error or attack.

Alert and Application Protocols: always two byte msg. First byte = severity (warning, fatal -> terminate connection). Second byte indicate preset error code. *Alert = Exception*

Pros: easily implementable
Cons: slower, needs dedicated port (443 for HTTPS), needs reliable transport (no UDP)

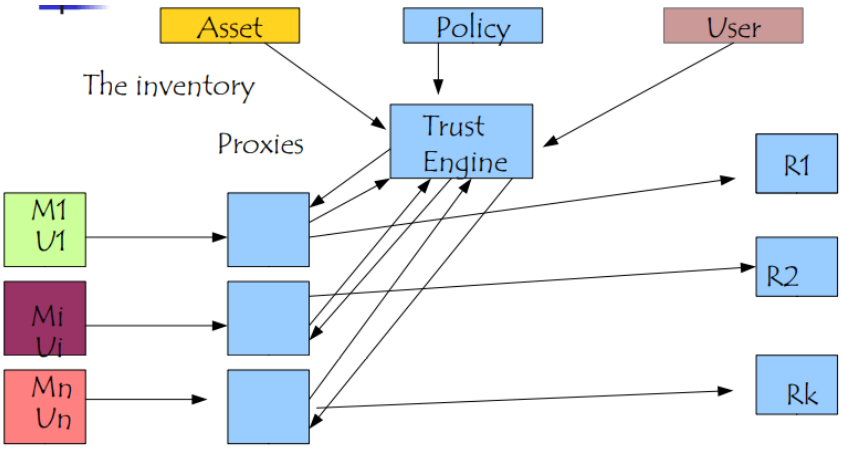
Example (vuln): Heartbleed It allowed attkrs unprecedented access to sensitive informations, caused by a flaw in OpenSSL. A heartbeat is an important part of TLS/SSL protocols: essentially, it's how two communication computers let each other know they're still connected even if the user isn't doing anything. Occasionally, one will send an encrypted piece of data *(heartbeat request)*, the other will reply with the same exact encrypted piece of data: the request includes info about its own length. Msg: "this is 40KB, repeat it back to me thx" -> server allocates 40KB memory buffer. OpenSSL wouldn't check if the request was actually as long as it claimed to be! If i'm actually 20KB and I ask 40, I get the next 20 as well! This is all due to a single line of code *memcpy(bp, pl, payload)*

Returning to VPNs, they won't solve the problem if the malware is already hosted in a node

- **Inventories**
  - **Asset**: for each machine, info on the os, IDS, patches applied
  - **User:** the role of each user, the resources it needs to access

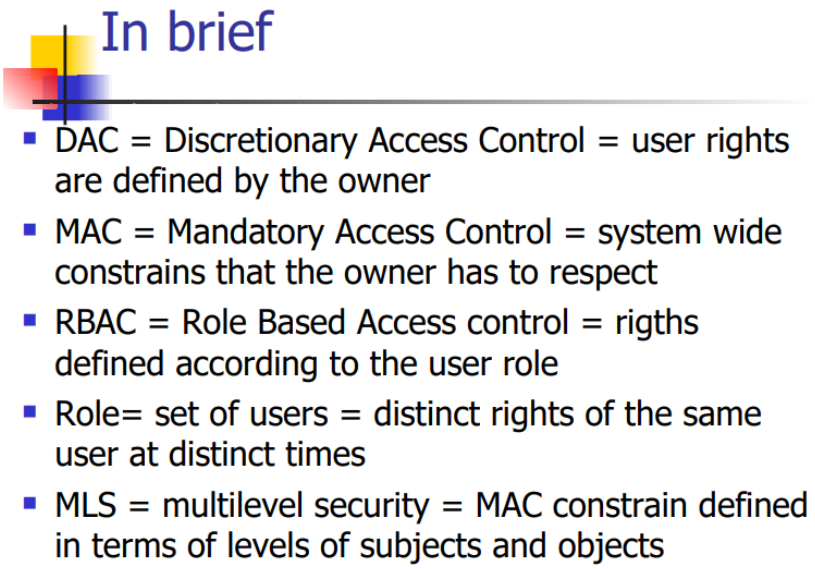
When A wants to access a resource R using a machine M, the engine computes a trust value: then the access is granted according to a trust threshold.

Zero-Trust (the future of VPN)



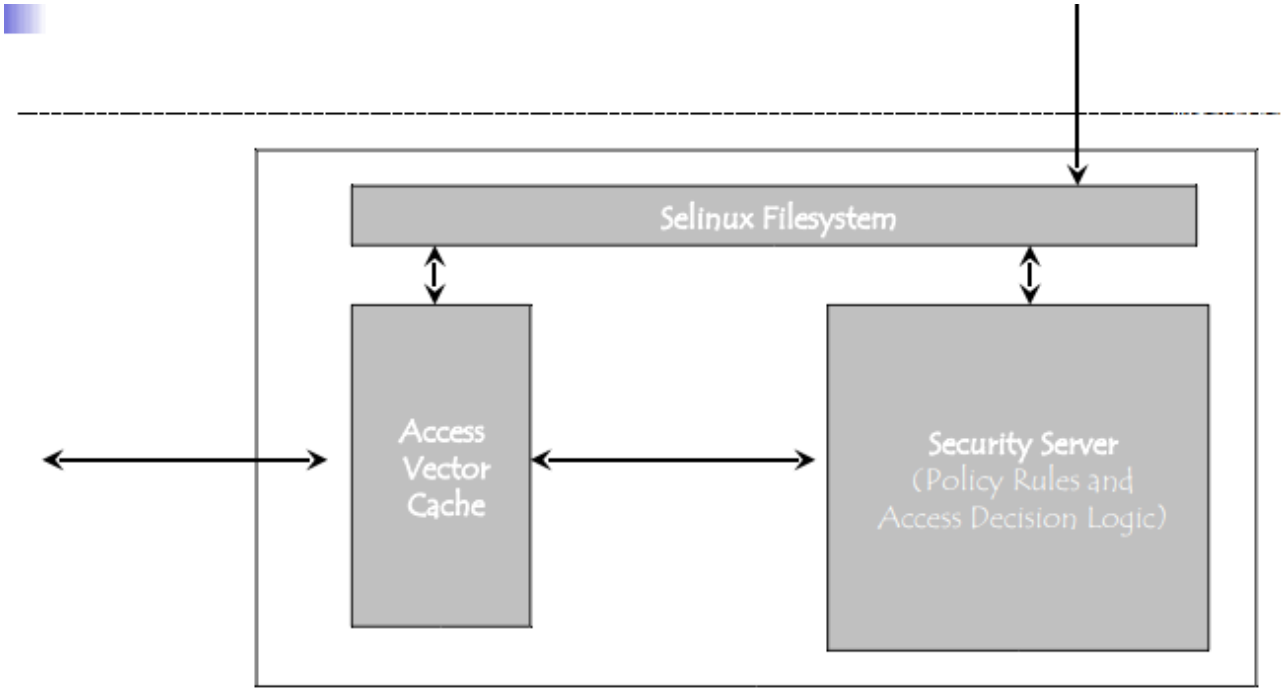
The proxy receives a request. It then transmits the request parameters and the certificates to the engine. If the engine grant access, the proxy creates an encrypted connection with the machine and interfaces the access to a resource. *The proxy can also act as a load balancer, but its main role is to enforce control.*

Again some slides about how security in lower levels are important more than on upper ones (see OS: Linux vs Enhanced Security Linux). Linux defines the user rights. SeLinux defines the rights (defined in terms of types, roles and levels [type1 can do OP_x on type2]) of each program and the program that each user can run while there's no root user, but the security policy is a configuration parameter.



LSM

- Kernel framework for security modules
- Provides a set of hooks to implement further security checks
- Usually placed after existing DAC checks and before resource access
- Implications? SELinux check is not called if the DAC fails
- Makes auditing difficult at times.



Webstone: create a load on web server by simulating multiple clients which can be thought of as users, web browsers that retrieve files from a web server.

AppArmor: supplements DAC (impossible to grant a process more privileges than it had at the beginning) in an easy way instead of configuring (hard) SELinux. It proactively protects the OS and apps from external and internal threats and even 0-day atks by enforcing a specific rule set.

Virtual Machines

They're not tied to physical machines, can run multiple VMs on a single desktop/server and offer multiple types of isolation. **Data isolation** (each VM is managed independently, different OS, different disks -> files/registry, different MAC addresses -> IP address, impossible data sharing), **faults isolation** (crashes are contained within a VM), **performance isolation** (guaranteed), **security isolation** (no assumptions on the SW running inside a VM).

They offer many advantages like “see without being seen”, because it’s difficult from within a computer but it’s easy on the host.

Currently: hybrid is arising, resilience *(“the capacity to recover quickly from difficulties”* – Oxford) is the new goal: to achieve it, identify key threats and assess their impact on critical cybersystems and functions, increase robustness, classify and prioritize critical services, set cyber-resilience goals and objectives for critical services. Any resilience requires some redundancy.

Virtualization

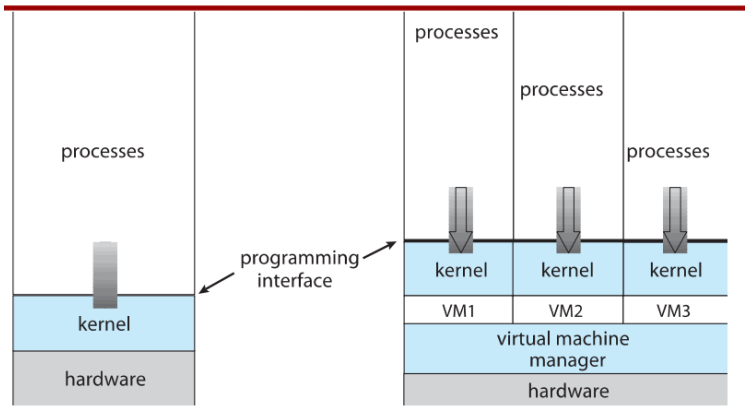
We’re starting to talk about the security of Cloud Computing. So there are different threat models, new attacks and therefore different countermeasures.

The first supporting technology we face is the virtualization. Its idea is to abstract HW of a single computer into several different execution environments. It’s somewhat similar to a *layered approach*, but layers here are virtual systems on which OS/apps can run. Once created by the VMM, it gets some resources assigned to it *(#cores, memory, network)* either dedicated, shared or a mix of both. When no longer needed the VM can be frozen or deleted -> *possible virtual machine sprawl (hard to track and manage VMs).*

There are several components:

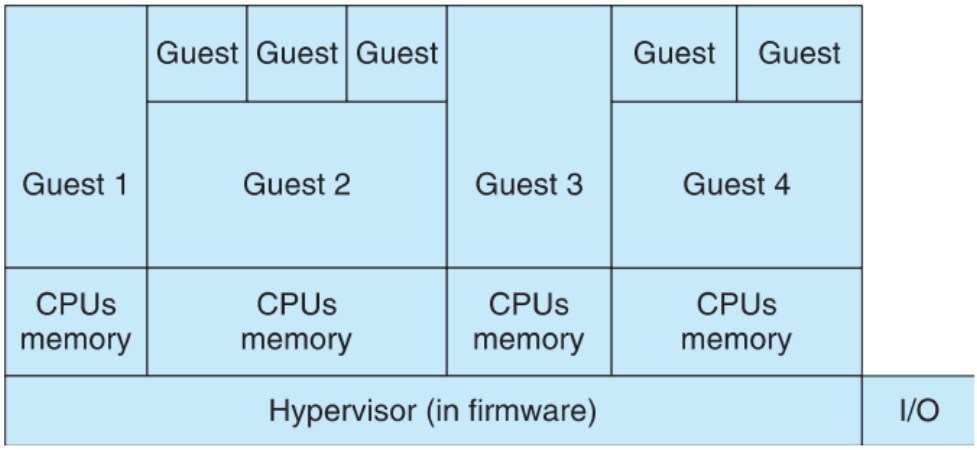
- **Host:** underlying HWW system
- **VM manager (hypervisor):** create and runs VMs by providing interface that is identical to the host *(not in paravirtualization)*
- **Guest:** process provided with virtual copy of the host *(usually an OS)*.

So, a non VM and a VM are represented below:

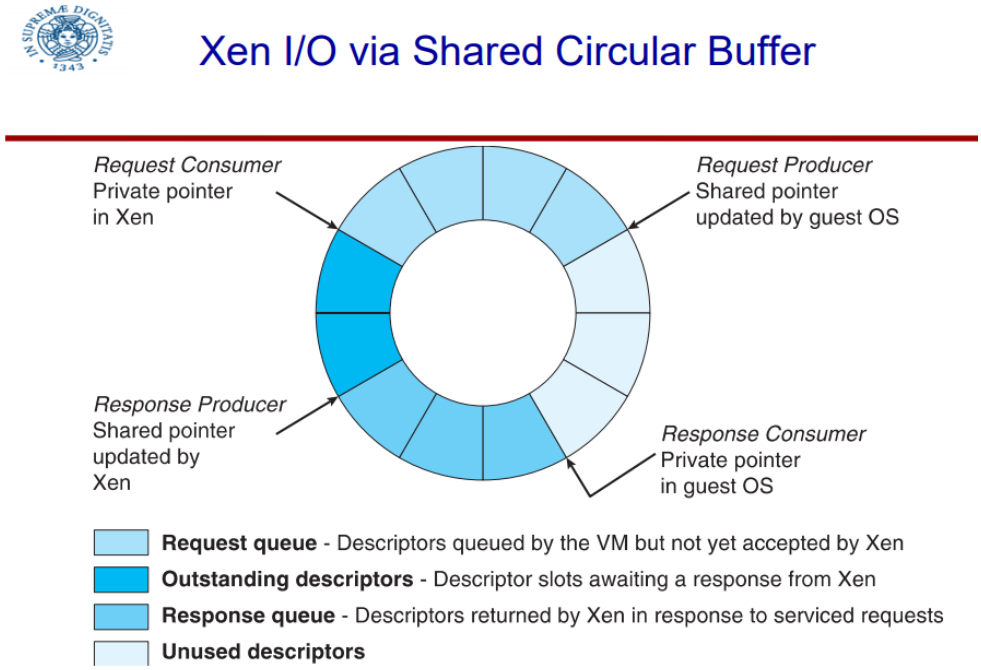


Implementation of different VMMs are possible, including:

- **Type 0:** HW-based solutions that provide support for virtual machine creation and management via firmware. *So the VMM is in the firmware and OS has nothing special to do.*
  It has the smallest feature set out of all the types and so each guest has its own dedicated HW *(so I/O is a challenge because it’s hard to have enough devices for each guest)*. For this reason, the VMM may implement a *control partition* running daemon that other guests communicate with when shared I/O is needed.



- **Type 1:** OS-like SW built to provide virtualization. It also includes general-purpose operating systems that provide standard functions as well as VMM functions. *Commonly found in datacenters due to the consolidation of multiple OSs and apps onto less HW, the ability to move guests between systems to balance performance and the possibility of snapshots and cloning.*
  Rather than providing a system call interface, it creates, runs *(in kernel mode)* and manages guest OSs and so they generally don't know they're running on a VM. This hypervisor has to manage snapshots and cloning.
- **Type 2:** applications that run on standard OS but provide VMM features to guest OS. The VMM is simply another process, run and managed by the host *(which doesn't even know there are VMMs on it, it's very little OS involving)*. They're usually poorer in performance because they can't fully exploit HW features, but pros are that they require no changes to host OS.
- **Paravirtualization:** a technique in which the guest OS is modified to work in cooperation with the VMM to optimize performance. *Remember that VMM isn't presenting an exact duplication of underlying HW.* Xen (leader in paravirtualized space) has clean and simple device abstractions for efficient I/O:



- **Programming-environment virtualization:** VMMs do not virtualize real HW but instead create an optimized virtual system
- **Emulators:** allow apps written for one HW environment to run on a different one *(e.g. different CPU)*

- **Application containment:** no virtualization at all, but it rather provides virtualization-like features by segregating apps from the OS, making them more secure and manageable.

Example: the Linux Containers (LXC) feature

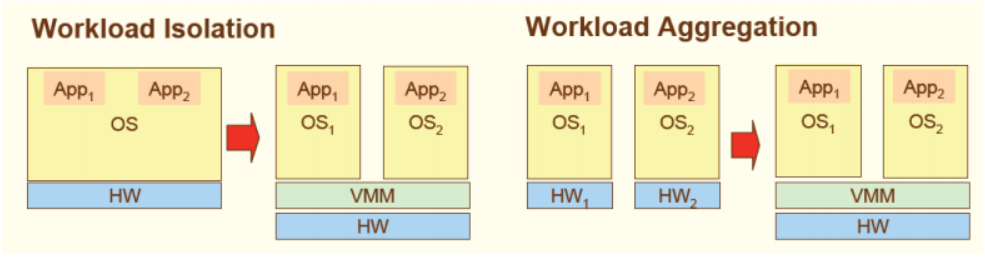
A lightweight virtualization mechanism that doesn't require you to setup a VM on an emulation of physical HW. It takes the *cgroups* resource management facilities as its basis and adds POSIX file capabilities to implement process and network isolation. There you can run many things, like a single app within a container with an isolated namespace, or a complete copy of the Linux OS in a container without the overhead of a lvl-2 hypervisor (e.g. virtualBox). This means that the container will share the kernel with the host system, and so its processes and file system will be completely visible from the host, while being on the container you can only see its file system and process space.

So it's a good idea to use **container-specific host OS** to reduce attack surfaces. It's a minimalist OS explicitly designed to only run containers, with everything else disabled and with read-only file systems.
Examples: CoreOS, RancherOS, ProjectAtomic, ubuntu core, Photon OS

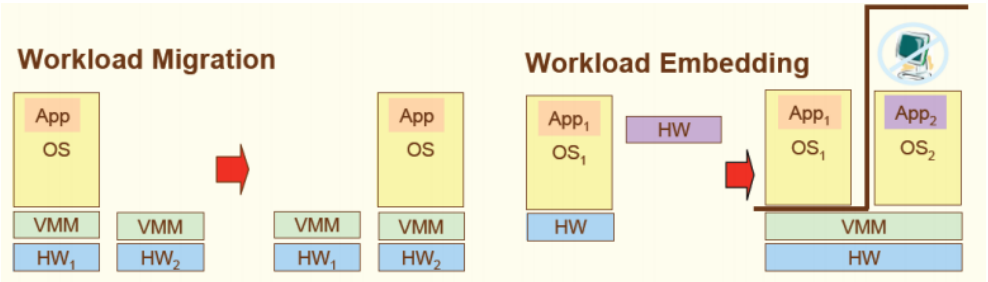
You may group containers on a single host iff they share the same purpose, sensitivity and threat posture. This grouping increases the complexity for an attacker who compromises one of the groups to expand to other groups, while also increasing the likelihood that compromises will be detected.

Adopt container-specific vulnerability management tools and processes for images to prevent compromises. Traditional tools aren't usually able to detect vulns within containers. A reporting and monitoring of the compliance state of each image would prevent noncompliant images from being run. The network traffic will be managed by the orchestrator such that packets will be opaques *(encrypted)* to security and management tools.

So VMs are good because they're protected from each other *(harder for a virus to spread)*, you can freeze/suspend/clone VMs, you can do a live migration without interrupting user access, run multiple and different OSs on a single machine, etc. All together, we're starting to see *Cloud Computing*: through APIs, programs tell cloud infrastructure to create new guests, VMs, etc.



Pros of Cloud Computing are the usual ones: efficiency (time multiplexing, ..), resource control, etc.

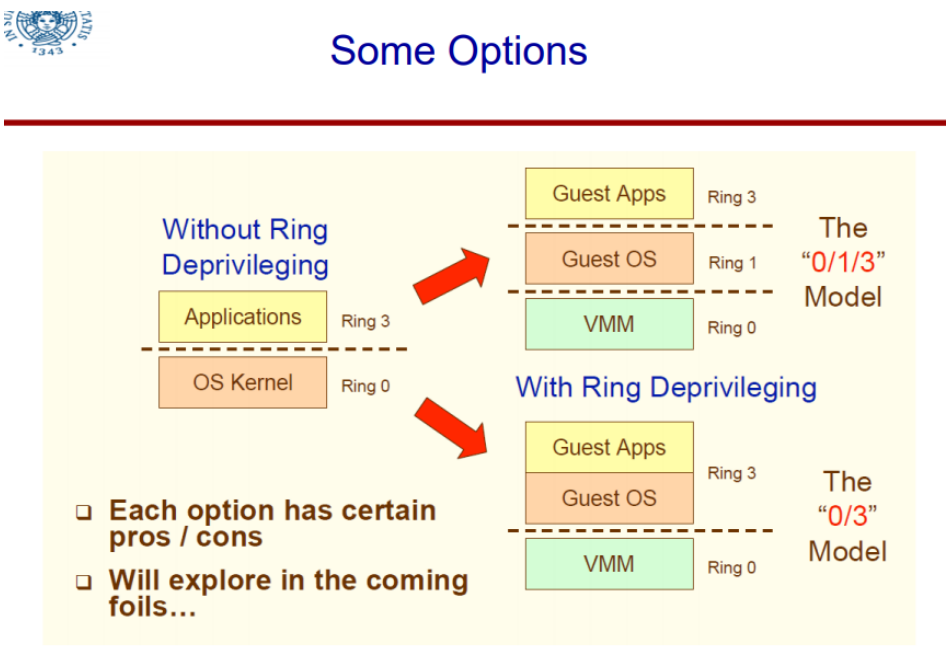


Assuming that the set of instruction of a physical macchine can be splitted into *(Popek and Golberg)*:

- **Privileged instructions:** trap if the processor is in user mode and don't if it's in system (supervisor) mode. *Achieve this by running guest OS deprivileged*
- **Control sensitive instructions:** attempt to change the configuration of resources in the system
- **Behavior sensitive instructions:** their behavior depends on the configuration of resources

Then the main analysis can be expressed by the following theorem:

"For any conventional third-generation computer, an effective VMM may be build if the set of sensitive instructions for that computer is a subset of the privileged ones"



Ring deprivileging: all guest SW should be run at a privilege lvl > 0. Privileged instructions generate faults *(so VMM runs in Ring-0 as a collection of fault handlers)*. The guest OS can't update the VMM.

The VMM interprets in SW privileged instructions that would be executed by an OS: any other instruction will be executed directly by the machine. There are 2 ways that a guest OS could be deprivileged: (1) it could run either at privilege lvl 1 *(called **0/1/3 model**)* or (2) it could run at privilege level 3 *(**0/3/3 model**)*.

Some challenges:

Ring aliasing may be a problem if the SW is run at a privilege lvl other than the privilege lvl for which it was written.

Excessive faulting: ring deprivileging interferes with the effectiveness of facilities that accelerate the delivery and handling of transitions to OS software.

Non-Trapping Instructions: some instructions don't fault when executed with insufficient privilege.

Interrupt virtualization: masking external interrupts for preventing their delivery when the OS isn't ready for them is a big challenge for the VMM design. The VMM must manage this because it prevents any guest OS to receive interrupts.

Access to Hidden State: some components of the processor state aren't represented in any SW-accessible register.

Ring compressions: two main mechanisms, segment limits and paging

To **Trace memory** means to protect cache tables in a transparent way to the guest. A **shadow page table** is used by the hypervisor to keep track of the state in which the guest "thinks" its page tables should be. *The guest can't be allowed access to the HW page tables because it'd mean total control of the machine.* So the hypervisor keeps the "real" mappings *guest virtual -> host physical* in the HW when the relevant guest is executing and a representation of the page tables that the guest think it's using "in the shadows". These tables are loaded into the MMU on a context switch: the VMM needs to keep its *V -> M* tables consistent with changes made by OS to its *V -> P* tables (memory tracing).

VMM may be at SW (very flexible) or HW (concise): if a lot of I/Os and process management is involved, SW prevails. Otherwise if there's a lot of system calls, HW prevails.

Cloud Computing

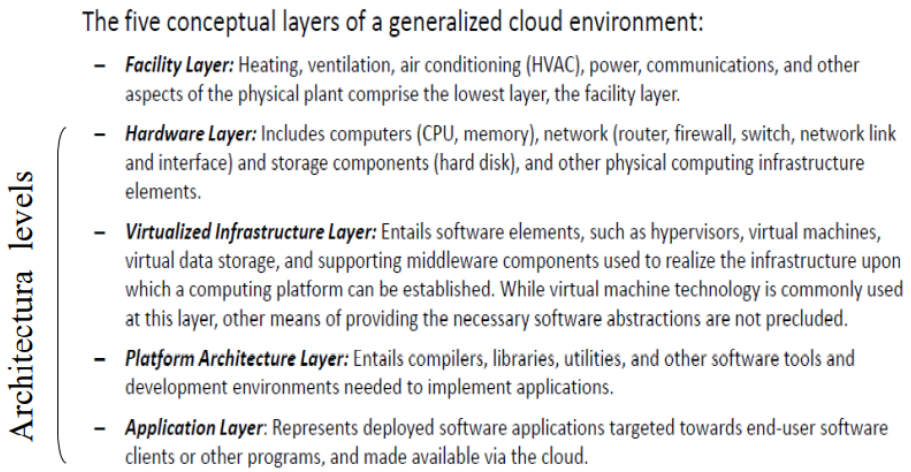
The cloud service model tells us how a service is erogated and which are the duties and rights of the service provider/service customer.

The cloud deployment model represents a specific type of cloud environment, primarily distinguished by ownership, size, and access. It defines where the infrastructure for the deployment resides and who has control over the infrastructure.

The two models are almost orthogonal.

Service *(through SLAs)*:

- **SaaS *(software as a service):*** *use apps the provider supplies over a network*
  Scalable, Multi-tenancy *(one app instance may be serving hundreds of companies)*, Metadata driven configurability
- **PaaS *(platform as a service):*** *deploy customer-created apps to a cloud.*
  Don't worry about provisioning servers. Facilities like DB management, security, etc.
- **IaaS *(infrastructure as a service):*** *rent processing, storage, network capacity, etc*
  In other words, this model is based on the same principles of virtualization: rather than running a virtual image on a partition on a physical server in a data center, the code runs on a VM created in the cloud.



A further division is:

- **Private Cloud:** intentionally limit access to resources to consumers belonging to the organization who owns the cloud. Also called *internal cloud* or *on-premise cloud*.
- **Public Cloud:** it provides an IT infrastructure in a 3rd-party physical data center that can be utilized to deliver services without having to be concerned with the underlying technical complexities. Also called *external cloud* or *multi-tenant cloud*.
- **Community Cloud:** it refers to special-purpose cloud computing environments shared and managed by a number of related organization participating in a common domain/market. Variations of this:
    - **Hybrid Cloud:** merge private and public cloud environments
    - **Dedicated Cloud:** environmennt hosted and managed off-premise or in public envs, but dedicated resources are provisioned solely for private use. Also known as *hosted cloud* or *virtual private cloud*).

Cloud computing is of course getting more and more popular every year because data centers are way more expensive in every possible way *(use 15% only of what is paid, pay full capacity to not underestimate accesses, etc)*.

Rule #1: utility services cost less even thought they cost more. When they're used they are expensive, but most of the time you're not using them.

Rule #2: on demand trumps forecasting. React instantaneously to unexpected demand, higher or lower.

Rule #3: the peak of the sum is never greater than the sum of the peaks. The reallocation of resources across many enterprises with distinct peak periods implies that a cloud needs to deploy less capacity.

Rule #4: aggregate demand is smoother than individual. Specifically, the coefficient of variation (ratio of the stdev to the mean) of a sum of random variables is always less or equal than to the one of each individual variable. Therefore, better economics

Rule #5: average unit costs are reduced by distributing fixed costs over more units of outputs.

Rule #6: Superiority in numbers is the most important factor in the result of a combat. Following the classic military strategies where numerical superiority is the key to winning battles, an enterprise IT would be overwhelmed by a DdoS atttack, whereas a large cloud service provides has the scale tto repel it.

Rule #7: space-time is a continuum. A real-time enterprise must respond to business conditions and opportunities faster than the competition. So the compute job must be elastic to trade off space and time *(e.g. a job may run on one server for 1000 hours or 1000 servers for one hour)*.

Rule #8: dispersion is the inverse square of latency

$$T = F + \frac{N}{\sqrt{n}} + \frac{P}{p}$$

Simply put, the total response time T for an interactive networked application in which a client application requests and receives an interactive response from a cloud application over a network is a function of three components. F is a fixed interval that can't be accelerated, N is the worst case round-trip latency for an environment with a single node, n is the number of (evenly-distributed) processing nodes, P is the time for the parallelizable portion of the application to run on one processor, and p is the number of processors.

For example, consider a search query requested via an end-user client. A time F is needed for client processing and other serial tasks; sharding and parallelization can reduce processing time via the P/p component; and replicating this service in n multiple physical locations can lower the network latency, reducing the $N/\sqrt{n}$ component. In fact, if Q processors are available for deployment, the optimum latency is reached when the number of nodes is $n = \sqrt[3]{(\frac{QN}{2P})^2}$.

Rule #9: never put all your eggs in one basket. No finite number of data centers can reach 100% reliability. To provide high availability services globally for latency-sensitive applications, there must be a few data centers in each region.

Rule #10: an object at rest tends to stay at rest.

Resuming…

Advantages

- Shifting public data to a external cloud reduces the exposure of the internal sensitive data
- Cloud homogeneity makes security auditing/testing simpler
- Clouds enable automated security management
- Redundancy / Disaster Recovery
- Dedicated Security Team
- Greater Investment in Security Infrastructure
- Hypervisor Protection Against Network Attacks

Challenges

- Trusting provider security model
- Customer inability to respond to audit findings
- Obtaining support for investigations
- Indirect administrator accountability
- Proprietary implementations cannot be examined
- Loss of physical control
- Legislation

All of this implies the existence of a new threat model, as we expected. Now you don't have to protect your resources anymore *(processor, memory, network)*, but you have to **protect your information**. The cloud provider will handle the physical assets, all you need to protect is virtual.

The attack surface has been *extended* (VMM itself and the browser used to interact with the cloud) and we must take this into account, because we know that the notion of surface allows us to evaluate the percentage of a system exposed to threat attacks. Of course, it depends on the type of cloud we're using *(private = good control, public = no control, community = acceptable)*.

An attacker that is successful can access a much larger amount of resources, know how, processing power than the typical one. **Note:** the cloud provider can be a new threat itself *(impossible to be sure that data stored has been erased, etc)*. That's why we define SLAs (include ONLY properties that can be measured!).

Let's now see a checklist of the most usual cloud vulnerabilities

- **Authentication Vulnerabilities:** insecure storage of cloud access credentials, insufficient roles, etc.
- **Resource Vulnerabilities:** over-provisioning, no resource capping, failures in resizing resources, etc.
- **Remote Access to Management interface:** compromise cloud infrastructure through weak auth.
- **Hypervisor itself:** exploiting him means exploit every VM, through *host-escape, VM-hopping, etc.*
- **Isolation vulnerabilities:** side channel attacks, shared storage, etc.
- **Weak / No Data Encryption in transit:** therefore also poor Encryption Key management.
- **Low Entropy for Random Number Generators**
- **No Control of Vulns Assessment Process:** restrictions on port scanning and vulns testing are an important vulnerability which places responsibility on the customer for securing elements of the infrastructure.
- **Internal (Cloud) Network Probing:** customers can perform port scans on others within internal net.
- **Co-Residence checks:** side-channel atks exploiting resource isolation allow attackers to determine which resources are shared by which customers.
- **Lack of Forensic readiness:** cloud should VM freeze etc, but several providers don't provide it!
- **Media Sanitization:** data destruction policies may be impossible to implement and so sensitive data may leak on shared tenancy of physical storage resources
- **Service Level Agreement:** clauses with conflicting promises to different stakeholders or with clauses from other providers
- **Audit or Certification not available to customers**
- **Certification schemes not adapted to cloud**
- **Storage of Data in multiple jurisdictions:** mirroring data for delivery by edge networks and redundant storage without real-time information available to the customer of where data is stored
- **Lack of information on jurisdictions:** data may be stored/processed where it's vulnerable to confiscations.
- **Lack of cloud security awareness:** some may not know about the loss of control on data, provider lock-in, exhausted resources problems etc when migrating into the cloud.
- **Lack of vetting processes (personal background checks):** there may be very high privilege roles within cloud providers
- **Unclear roles and responsibilities**
- **Poor enforcement of role definitions:** a failure to segregate roles may lead to roles to be too privileged and make systems vulnerable
- **Need-To-Know principle not applied:** parties shouldn't be given unnecessary access to data
- **Inadequate security procedures:** lack of electromagnetic shielding, perimeter controls, etc.

Cloud Computing Vulnerability

How can a user detect that the application is running on a VM as a first step to attack the VM itself? Or also obfuscate malware in a testing environment. Transparency VS Compatibility

Detect VM artifacts in processes, in File System or in Registry. Look for VME artifacts in memory, or VME-specific virtual HW / processor instructions. (e.g. references to "VMware" in memory, critical OS structures located in different places, etc.)

One particular memory difference is the location of the Interrupt Descriptor Table (IDT). On Host machines it's typically low in memory, while on Guest machines it's typically higher in memory. It can't be the same, because the processor has a register pointing to it (IDTR)

Example: The Red Pill (2004) *http://invisiblethings.org/index.html#redpill*

Reliably detect VM usage without looking for file system artifacts. It runs a single machine instruction, SIDT (Store Interrupt Descriptor Table), which can be run in user mode and takes the location of the IDTR and stores it in memory where it's analyzed. On Vmware guest machines, IDT is typically locateed at 0xffXXXXXX while on virtualPC guests, it's located at 0xe8XXXXXX. On Host OS, lower (windows 0x80ffffff, Linux 0xc0ffffff). So if the first byte returned by SIDT is > 0xd0 -> VM, else real machine.

Otherwise, some VMEs introduce extra instructions that can be brought into the processor to see if it rejects them or it can handle them. Many practical examples can be found in the slides.

The main problem is still the same: **compatibility is not transparency**.

This because compatibility != isolation and is not security.

An important feature of virtualization is the **loss of monotonicity**. It causes a server's history to stop being a straight line. Instead, it becomes a graph, where branches are made on replication and copy operations and a previoous state can be reached when a restore is performed -> Data can't be deleted easily *(many copies).*

Furthermore: web attacks!

Sql Injections (in 2019, it's still 2/3 of the web application attacks……)

Remote/Local File Inclusion (RFI/LFI): vulnerability found in poorly-written web apps, when the user is allowed to submit input into files or upload files to the server. **LFI** allows an atker to read (and maybe execute) files on the victim machine: dangerous because if the web server is running with high privileges, the atker may gain access to sensitive informations. **RFI** instead are easier to exploit but less common, it lets the atker to execute stuff hosted on their own machine.

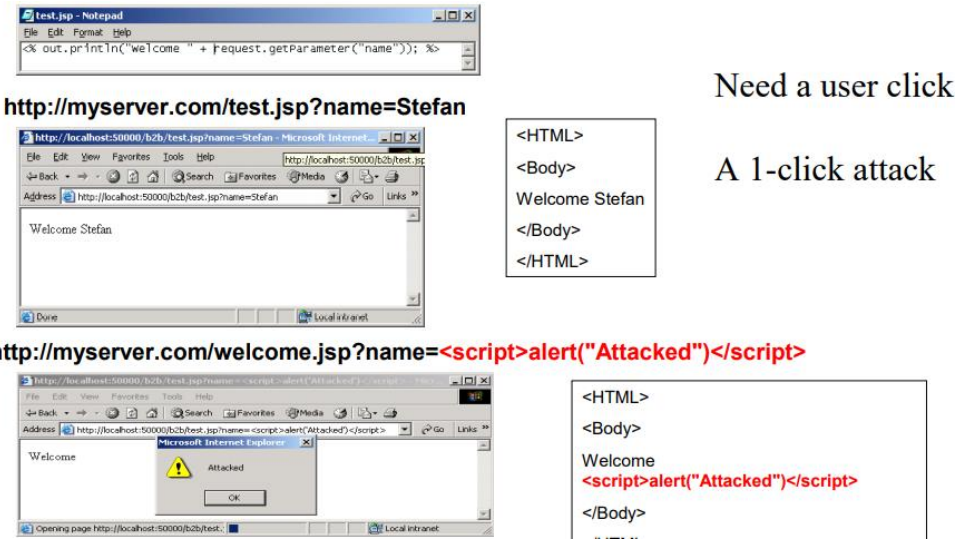
Cross-Site Scripting (XSS): web browsers can execute commands (embedded in HTML pages, usually JS). Cross-Site means "foreign script sent via server to client": the atker makes the web-server deliver malicious script code to the client and this will be executed in the client's web browser with the trust of the server. It's used to steal access credeentials, DdoS, or simply to execute any command at the client machine. The 3 conditions needed for XSS arre: **the web app accepts user input**, then **the input is used to create dynamic content** *(dynamic web pages)* [any web app does both of this] and finally **input isn't sufficiently validated**. *Note that the browser has no way of recognizing that the code isn't legitimate and malicious.*

How does this relate to cloud? Imagine the atker script retrieves the authentication cookie that provides access to a website W, and then posts the cookie to a web address known to the atker. The atker can spoof the legitimate user's identity and gain illegal access to W. If W is the interface to access a cloud

architecture, the atker gains access to all the cloud resources the client can access. At this point, the cloud provider can't do much.

There are 3 kinds of XSS:

- **reflective attack** *(a target attack for a single user, spear phishing)*



- **stored attack** *(a mass attack to a number of users)*

  User input is read from a request and stored in raw form (Database, File). Example comment in a blog. Great website<script src="htt//xss.xss/xss.js"></script>

- **Dom based attack** *(changes the execution environment)*

  It occurs when the DOM environment is being changed, but the client-side code doesn't. When the DOM env is being modified in the victim's browser, the client side code executes differently. Consider http:…/a.htm?default=1 "default" is a parameter and 1 its value. XSS DOM ATK would send a script as the parameter. It's similar to the "reflective" but the server doesn't play a role: the fault is within client-side JS code *(caused by output in HTML context or JS eval() injection)*, and it also may not need a click!

So, all depends on input validation. **Check if the input is what you expect, don't check for "bad input"!!** (black lists are never complete). White list testing is better (regular expressions). Furthermore, to help prevent XSS attacks, an application needs to ensure that all variable output in a page is encoded before being returned to the end user. Encoding variable output substitutes HTML markup with alternate representations called entities. The browser displays the entities but doesn't run them: *for example, <script> gets converted to <script>.* When an entity is encountered, it'll be converted back to HTML but not run.
There are fields where this isn't possible, like when constructing URLs from input *(e.g. redirections)*, or meta refreshes, HREF, SRC, … . There are fields instead where this is not sufficient, like when generating JS from input or when used in script enabled HTML tag attributes:

htmlencode("javascript:alert('Hello')") = javascript:alert('Hello')

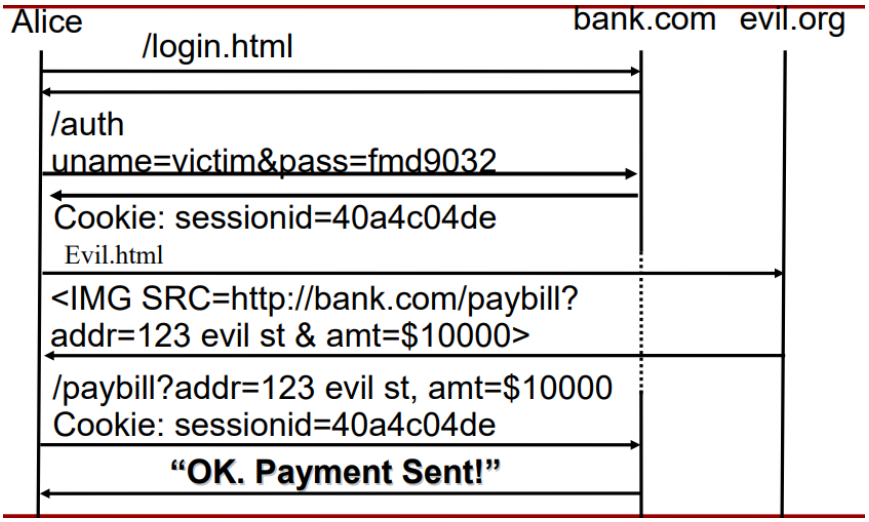
Finally, httpOnly Cookies *(Facebook, Google)* complicates attacks on cookies. The httpOnly flag tells the browser that the cookie should only be accessed by the server, otherwise strictly forbidden. It prevents disclosure of cookie via DOM access. But cookies are sent in each http request, passwords may still be stolen via XSS "secure" cookies, etc.

Cross-Site Request Forgery (CSRF)

Also known as *one-click atk* or *session-riding*. A malicious site instructs a victim's browser to send a request to an honest site as if the request was part of the victim's interaction with the honest site. *E.g. http//bank/transfer.do?acct=ME&amount=10000 when the user is authenticated to the e-banking site.* They're effective in a lot of situations, like: the victim has an active session on the target site, the victim is authenticated via http auth on the target site or the victim is on the same local network as the target site. During the attack, the attacker disrupts the integrity of the session *user ←→ website* by injecting network requests via the user's browser. The browser's security policy allows websites to send http requests to any network address: this policy allows an atker that controls content not otherwise under his or her control to **read browser state** *(cookie, certificate)*, **write browser state** *(set cookie)* or **network connectivity** *(behind firewall)*.

Example:

Alice is using a good website, bank.com, and is logged with a cookie. At the same time, she's visiting also evil.org.



Malicious site can't read info (due to same-origin policy) but can make **write** requests to out app and can so still cause damage *(gain full read/write access on Alice's account)*!

Who should worry about XSRF? Apps with user info and profiles (Facebook, etc), apps that do financial transactions for users and any app that stores user data (CLOUDS)

➔ Same Origin Policy

It's an important security measure in browsers for client-side scripting. "Scripts can only access properties associated with documents from the same origin". The origin reflects hostname, protocol and port.

http != https .com/ != .com:81/ abc.com != extranet.abc.com

This restricts network capabilities (important exception: cross-domain links in the DOM are allowed), so scripts can't read DOM elements from another domain. Although, documents of different origins can still interact:

- **links to other documents:**

  Click here!

  Links are loaded in the browser (with or without user interaction) possibly using cached credentials

- **using iframes:**

  <iframe style="display: none;" src="domain/path"></iframe>

Link is loaded in the browser w/o user interaction, but in a different origin domain

- **using external scripts:**

  <script src="domain/path"></script>

  The origin domain seems to be "domain", however the script is evaluated in the context of the enclosing page. This results in a script that can inspect the properties of the enclosing page and the enclosing page can define the valuation environment for the script.

- **submitting requests:**

```
<form name="myform" method="POST" action="http://mydomain.com/process">
    <input type="hidden" name="newPassword" value="31337"/>
        …
</form>
<script>
    document.myform.submit();
</script>
```

  Form is hidden and automatically submitted by the browser, using cached credentials. The form is submitted as if the user has clicked the submit button in the form

- **via the Image object:**

```
<script>
var myImg = new Image();
myImg.src = http..bank/xfer?from=1234&to=5678&amount=1000
</script>
```

- **via document.* properties:**

  document.location = http.. bank/xfer?from=1234&to=5678&amount=1000

- **redirecting via the meta directive:**

  <meta http-equiv="refresh" content="0; URL=bank/xfer?from=1234&to=5678&amou.."/>

- **via URLs in style/CSS:**

```
body { background: url('http..yourbank.com/xfer') no-repeat top }
<p style="background:url('http..yourbank.com/xfer');">Text</p>
```

- **using proxies, Yahoo pipes:**

  <LINK href="http..yourbank.com" rel="stylesheet" type"text/css">

To prevent XSRF, inspecting referer headers *(can be forged or blanked)* or using web app firewall *(request looks authentic to bank.com)* aren't enough. So the correct ways are validation via user-provided secret *(ask for current password for important transaction)* and validation via "action token" *(add special tokens to "genuine" forms to distinguish them from "forged" forms).*

Slides then end with a small talk on 0-clicks attacks, which are, obviously, attacks that don't require any action from the user.

Cartography attack & information leakage

A system where legal user and attackers share the same architecture is the target of new attacks that discover and monitor the flows of information *(among VMs, apps, platforms, ..)*. **Cartography =** how VMs are mapped.

Example: Amazon EC2

To use it, you simply select a pre-configured templated image to get up and running immediately *(or create an Amazon Machine Image [AMI])*. Configure security and network access on your amazon EC2 instance, then choose which instance type(s) and OS you want. You can then start/terminate/monitor as many instances of your AMI as needed, using the web service APIs or the variety of management tools provided. Amazon Elastic Block Store (EBS) offers persistent storage for EC2 instances and these volumes persists independently from the life of an instance. An elastic IP address approach is used, designed for dynamic cloud computing: the Elastic IP address is associated with your account not on a particular instance and you control it until you explicitly release it.

Strategy for cartography:

1. **Map** the cloud infrastructure to find where the target is located.
2. Use various **heuristics** to determine co-residency of two VMs
3. Launch **probe VMs** trying to be co-resident with target VMs
4. Exploit **cross-VM leakage** to gather info about target

Tools for network probing:

- **nmap:** security scanning and finderprinting to discover hosts and services on a computer network nodes to create a network "map"
- **hping:** packet generator and analyzer for TCP/IP
- **wget:** program that retrieves content from web servers.

External probes = originated from outside EC2, while internal are originated from EC2. Amazon EC2's own DNS to map dns names to IPs.

Each instance has one internal IP and one external IP, both static. Example:
External IP: 75.101.210.100
External Name: ec2-75-101-210-100.computer-1.amazonaws.com
Internal IP: 10.252.146.52
Internal Name: domU-12-31-38-00-8D-C6.computer-1.internal
Reverse engineering thee VM placement schemes provides useful heuristics about EC2's strategy.

For **Task 1:** we identified four distinct IP address prefixes, /16, /17, /18 and /19 as being associated with EC2. The last 3 contained public IPs observed as assigned to EC2 instances. Nmap has been used to discover IP public addresses by a scanning on port 80 and port 443. Via an appropriate DNS lookup from within EC2, we translated each public IP address that responded to either the port 80/433 scan into an internal EC2 address. **Results of the analysis:** *same instance type within the same zone = similar IP regions.*

For **Task 2: determining co-residence** (check to determine if a given VM is placed in the same physical machine as another VM). It's a network based check: instances are likely co-resident if they have matching Dom0 IP address, small packet round-trip times or numerically close internal IP addresses *(e.g. within 7)*. An instance owner can determine its Dom0 IP from the first hop on any route out from the instance. One can

determine an uncontrolled instance's Dom0 IP by performing a TCP SYN traceroute to it (on some open port) from another instance and inspecting the last hopNo false positives found during experiments.
For any pair A,B at first A probes B (control A) and then B probes A (control B).

Task 3: making a probe VM co-resident with target VM. We could go for a brute force scheme, launching many probes instances in the same area, but success rate is 8.4%. A smarter strategy is to **utilize locality**. The main idea is that VM instances launched right after target are likely to be co-resident with the target. This leads to 40% success rate due to strong locality in the allocation of instances!

Task 4, gather leaked information. Now that the VM is co-resident with the target, it can gather information via side channels (cooperating/unwwilling VM) […]
Side channel attack: is any atk based on information gained from the physical implementation of a cryptosystem rather than from brute force or algorithm weaknesses as in cryptoanalysis. It has many classifications *(timing, power, electromagnetic radiations, acoustic, cache, differential faults)*. Think about the CPU that contains small fast memory cache shared by all applications. If only the Atker accesses the memory, it's served from the fast cache. If the Victim also accesses memory, the cache fills up and the Atker can notice a slow-down, so the Atker can deduce the memory access patterns of the Victim which is sensitive information! *(Victim performing RSA/AES decryption, the access patterns are determined by the secret key -> get AES secret in 65ms)*.

[…] or perform DoS.
DoS: *two VMs, one creating a backup, the other executing "dd if=/dev/zero of=testfile bs=1M count=15000 eg" creates a 15G file with zeroes.*

If VMs are separated and secure, the best the attacker can do is to gather information, like measuring the latency of cache loads and use that to determine co-residence, traffic rates and keystroke timing.

The leaking attack can happen through a covert channel *(if i print at 10, it's 1, 0 otherwise)*: how to find a good one? Find a place where random data is being transmitted naturally and replace that random data with your own which is actually an encrypted message *(e.g. padding of several TCP segment headers)*. It's useful if you don't want your association with a node too be known *(so it's also used for accessing forbidden material and malicious activities)*. Otherwise just use another node to relay information for you *(heavy attack -> transmit 1, no requests -> transmit 0)*.

It can also happen through a shared CPU: executing or not in a given interval transmits a bit.

Our main findings:

- **Detailed user information is being leaked out from several high-profile web applications**
  - **personal health data, family income, investment details, search queries**
- **The root causes are some fundamental characteristics in today's web app**
  - **stateful communication,**
  - **low entropy input (a few input with distinct probabilities)**
  - **significant traffic distinctions (low density of observable values).**
- **Defense is non-trivial**
  - **effective defense needs to be application specific.**
  - **calls for a disciplined web programming methodology.**

Ajax many requests, stateful, very popular ☹

This leads to **Traffic Analysis Prevention (TAP)**. It'll of course cost in efficiency/performance and it limits the number of possible senders/recipients.

- **Mixing:** a packet travels from source to destination through several intermediate nodes *(mix nodes, onion routing)*. User encrypts the message to be sent. Each mix node collects packets and releases them in some different *(random)* fashion. The longer a node can hold and collect msgs, the better the security. Mixing helps to hide sender-recipient relationship by masking the route taken
- **Dummy msg:** injecting dummy packets into the network *(a.k.a. padding)*. Tradeoff: dummy messages may reduce amount of time real packets are saved up in mixed nodes, if dummy msgs are used in conjuction with a mixed network, but ofc increase network overhead.
- **Routing:** altering routes that packets travel to make traffic following difficult *(varying #hops)*. Like onion routing, each intermediate node only knows about previous and next hope, encryption/decryption at each node alters msg appearance.

In general TAP reduces the ability of adversaries, but these methods typically result in high overhead.

Example: Sidebuster

Memory as a Service attack & POR

Not only "software as a service", but also storage is becoming a more common business model where a client pays a server to store a file F. The file won't be retrieved, so the client can be sure that the server still has it (within an agreed response time) by retaining only metadata.

Proof of Retrievability **(or Posses, POR/POP)** provides some assurance that a party possesses a file, without actually retrieving it. **Objective:** provide "early warning" of deletion, corruption or other failures.

Block approach

- The file is splitted into d blocks at upload
- We check whether some blocks is still there
- The probability of non detecting that some block have been erased (an eraser) is

$$P_{esc} = \left(1 - \frac{m}{d}\right)^r$$

where

- r is the number of blocks we control
- m/d is the percentage of blocks that have been erased
- 1-m/d is the probability of selecting one block that has not been erased

Other schemes based upon homomorphic encryption allow anyone to check that the server stores the file ("challenge response MAC" was discussed in slide, N/R).

Homorphic encryption = Holy gray of encryption

Let R and S be sets and E an encryption $R \rightarrow S$
E is
- **Additively homomorphic if**
  E(a+b)=PLUS(E(a), E(b))
- **Multiplicatively homomorphic if**
  E(a×b)=MULT(E(a), E(b))
- **Mixed-multiplicatively homomorphic**
  E(*xy*)=Mixed-mult(*E*(*x*),*y*)
- *fully homomorphic if* there are no limitations on manipulations

The data + computation is on the provider side. Inputs are encrypted by the client, while outputs are transmitted to the client that decrypts it. This is a no trivial solution, because the provider executes most computations to prevent cases where the data is transmitted to the client, the client decrypts the data to then compute the results and encrypt, and the results are transmitted to the provider.

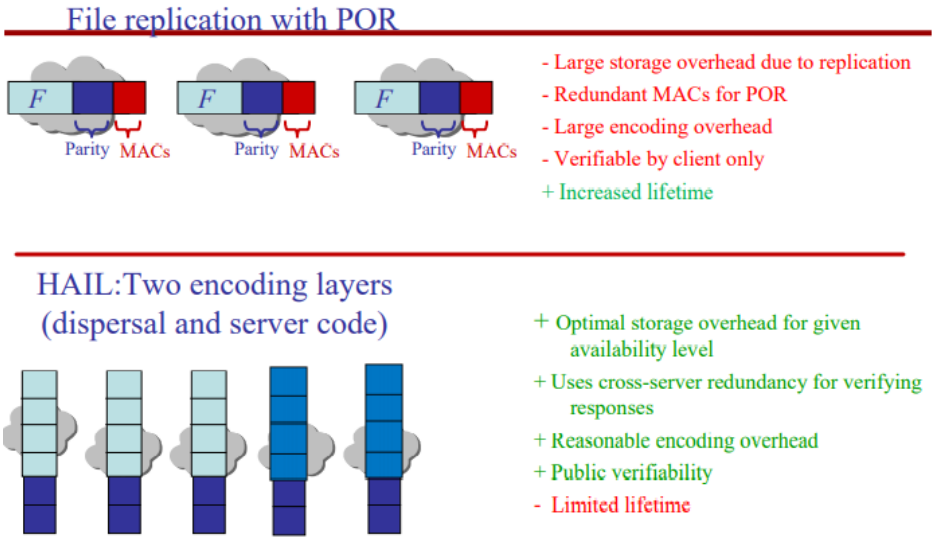
Another approach are **sentinels**, randomly constructed check values. F' = F encryption + emdeded sentinel, F is encrypted so that sentinels can't be discovered. The verification phase V specifies the positions of some sentinels in F' and asks the archive to return the corresponding sentinel values. About security, since F is encrypted and sentinels are randomly valued, the archive can't feasibly distinguished a priori between sentinels and portions of the original F. So, if the provider deletes/modifies a part of F', it'll w.h.p. also change a fraction of sentinels, and if V requests and verifies enough sentinels, V can detect whether the provider has erased or altered a substantial fraction of F'. **Individual sentinels are only one-time verifiable.**

The sentinel POR scheme has a curious feature that the sentinels and protocol msgs are independent of the file whose possession is being proved.

When there are more providers (imagine a RAID in a cloud environment), you could POR challenge-response each one of them. This is till vulnerable to small-corruption attacks *(once corruption exceeds the error correction rate of server code)*

HAIL goals

- Resilience against cloud provider failure and temporary unavailability
- A mobile adversary capable of progressively attacking storage providers and, in principle, ultimately corrupting all providers at different times.
- Use multiple cloud providers to construct a reliable cloud storage service out of unreliable components
- RAID (Reliable Array of Inexpensive Disks) for cloud storage under adversarial model
- Provide clients or third party auditing capabilities
- Efficient proofs of file availability by interacting with cloud providers
- Test-And-Redistribute the client uses PORs to detect file corruption and trigger reallocation of resources when needed
- On detecting a fault in a given server via challenge-response, the client recovers the corrupted shares from cross-server redundancy



But we can also have the **inverse problem: how can you be sure that data in the cloud has been erased?** In general, you can't, but there are solutions when data has been created outside and then stored in clouds. This leads to *retroactive attack on archived data*. This problem leaves us with **two huge challenges** for privacy. The first is that **data lives forever**, the second that **retroactive disclosure of both data and user keys** **has become commonplace**.

- **Self-Destructing Data Model**
  Until timeout, users can read original msgs. After timeout, all copies become permanently unreadable, even for an attacker who obtains an archived copy AND user keys. This will happen without requiring explicit delete actions by either the user or the service and without having to trust any centralized service *(which could have a backdoor agreement)*.

Traditional solutions *(PGP, centralized data mngmgnt services, forward-secure encryuption)* aren't sufficient for self-destructing data goals, so we'll try something completely new: **leverage P2P systems**.

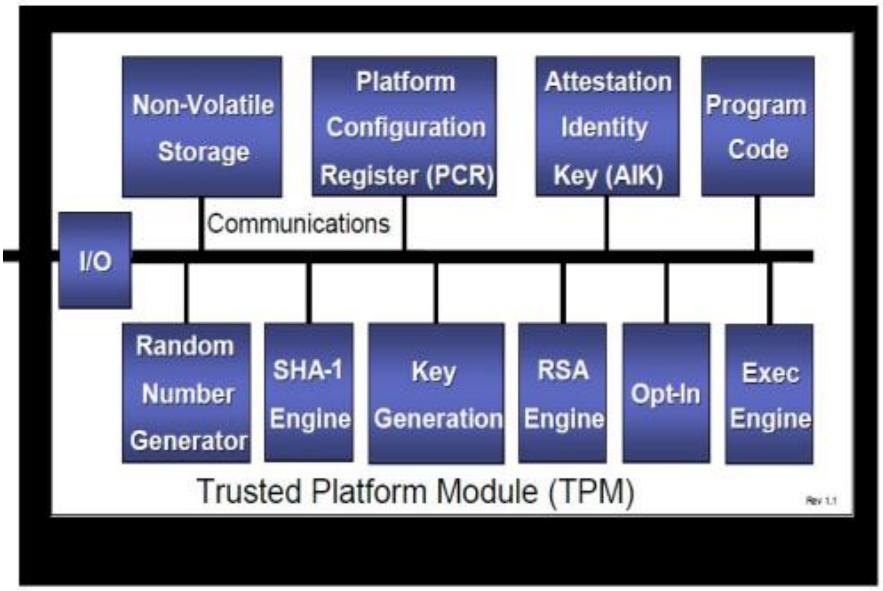
As usual, P2P classic properties: huge scale, geographic distribution, decentralization and network evolution, and this leads to actual DHTs.

Example: Vanish *http://vanish.cs.washington.edu/*
Example (Vanishing mails): Pluto Mail

Attestation

How to detect if an attacker (or provider) has updated the SW layers of the cloud system (violate any SLA)? The problemy in IaaS is that it's like a big black box, nothing inside the cloud is visible to the clients. Even an honest cloud provider can hire malicious sys admins who can tamper with the VMs and violate **C** and **I**. Asking the cloud provider to allow auditing of the cloud by the client would just result to provider not opening their system. So the only left approach is to ask the cloud provider for unforgeable proof / attestation.
Any solution to attest the integrity of any object, comparing the measurements against a stored one. So we're now trusting the measurements/measuring components.. **Trust problem can't be solved by recursion** *(measuring the measurer means trusting the root of the measurement system!)*.
In computer science, one solution for the root of trust is a **trustable anti-tamper HW component.**

Trusted Platform Module (TPM) is an inexpensive unit included in the chips of most laptops. It can be the building block for a trusted computing base, because keys never leave the module.



The PCR can contain hashes of system configurations.

TPM Module

- **Input/Output:** manages information flow over the communication bus. It performs protocol encoding/decoding for communication over external and internal buses. It routes msgs to appropriate components. It enforces access policies associated with the Opt-In component as well as other TPM functions requiring access control
- **Non-Volatile Storage:** It stores the Endorsement Key *(a private [RSA] key that identifies the chip)*, the Storage Root Key, owner authorization data and persistent flags. At least 16 PCRs have to be implemented in either volatile or non-volatile storage, because registers 0-7 are reserved for TPM while 8-15 are available for OS and application use. They're reset at system start or whenever the platform loses power.
- **Attestation Identity Keys:** it must be persistent, but it's recommended that AIK keys should be stored in persistent external storage (outside of the TPM) to speed up computation.
- **Program Code:** it contains firmware for measuring platform devices. Logically, this is the Core Root of Trust for Measurements (CRTM), that is contained in the TPM, but implementation decision may require it to be located in other firmware.
- **Random Number Generator:** A true random-bit generator to seed random number generation. It's used for key generation, nonce creation and to strengthen pass phrase entropy.
- **Sha-1 Engine:** A msg digest engine to compute signatures, create key Blobs and for general purpose use
- **TSA Key Generation:** YCG standardizes the RSA algorithm for use in TPM modules. The RSAKG engine creates *signin keys* and *storage keys*.
- **RSA engine:** it's used for signing with *signing keys*, encryption/decryption with *storage keys*, and decryption with the EK.
- **Opt-In**: this component implements TCG (Trusted Computing Group) policy requiring TPM modules are shippend in the state the customer desires. This ranges from **disabled** and **deactivated** to **fully enabled**, ensuring disabling operations are applied to other TPM components as needed.
- **Execution Engine**: it runs program code, it performs TPM initialization and measurement taking measurement *(compute an hash of the object to be measured)*

TCG requires of course that the TPM should be physically protected from tampering. These mechanisms are intended to *resist* tampering, but tamper evidence measures should enable *detection* of tampering upon physical inspection.

Non-Volatile Storage

These never leave the TPM

1. **Endorsement Key:** created at manufacturing time, can't be changed and is used for attestation. TPM operations signin pieces of data use the EK to allow other components to verify that the data can be trusted: to sign a piece of data, a private key encrypts a small piece of information. The signature can be verified by using the paired public key to decrypt that same piace of data.
2. **Storage Root Key (SRK):** it's embedded in the TPM security hardware. It's used to protect TPM keys created by applications, so that these keys can't be used without the TPM. This is created when the TPM is used, so it can be cleared and new ones can be created.
3. **Owner Password and persistent flags**

Platform Configuration Registers (PCR)

There are a lots of PCR registers on the chip (min 16) and their contents is a 20-byte SHA-1 digest (+junk). Initialized at a default value *(0)* at boot time, it has *TPM_SaveState* and *TPM_Startup(STATE)* to restore PCR values. To update the n-th PCR register, *TPM_Extend(n,D)* and to read *TPM_PctRead(n)*.

TPM_TakeOwnership(OwnerPassword, …) creates a 2048-bit RSA Storage Root Key on TPM: it can't be run again without *OwnerPassword* and it's done once by the IT department. *TPM_CreateWrapKey* and *TPM_LoadKey* for creating RSA keys on TPM protected by SRK. Encrypting works through *TPM_Seal(..)* that returns an encrypted blob, that can be decrypted by *TPM_Unseal*.

Anyway, through physical access problems may still rise *(sending TPM_Init on LPC bus allows arbitrary values to be loaded onto PCR, disabling TPM until after boot, etc)*. A better root of trust is the **Dynamic Root of Trust Measurement (DRTM)**: to check the integrity of a VMM it atomically resets CPU, reset PCR 17 to 0, loads given Secure Loader (SL) code into l-cache, extend PCR 17 with SL and jump to SL. This way, BIOS boot loader is no longer the root of trust.

Attestation: prove to remote party what SW is running on my machine. *E.g. bank allows money transfer iff customer's machine runs "up-to-date" OS patches, Quake playerss can join a Quake network only if their Quake client is unmodified, etc.*

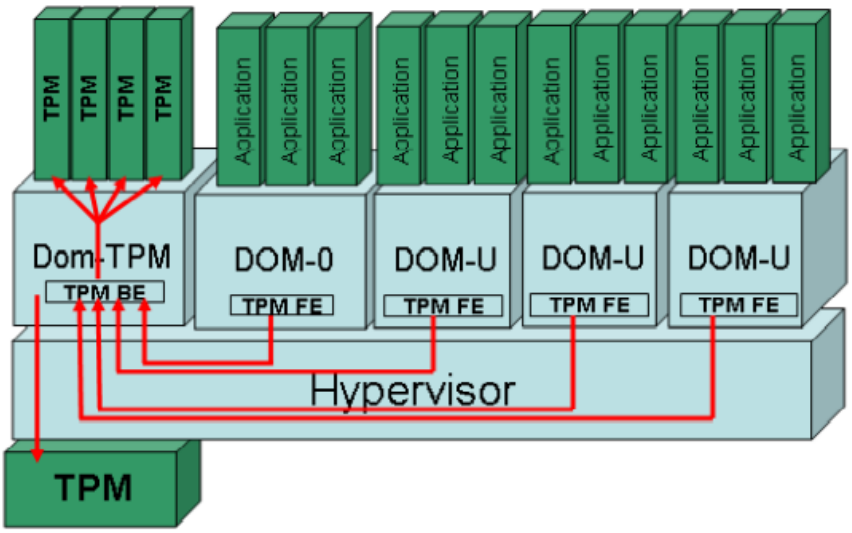
How it works (recall EK private key on TPM)

1. **Create Attestation Identity Key (AIK)**: AIK private key only known to TPM, while the public certificate is issued only if EK cert is valid.
2. **Sign PCR values (after boot)**: call *TPM_Quote* that returns signed data and signature

The security of the attestation report relies on the AIK certified by the Trusted Third Party *(a privacy CA)* on the basis of the EK, so **the root of trust for reporting (RTR) is the EK**.

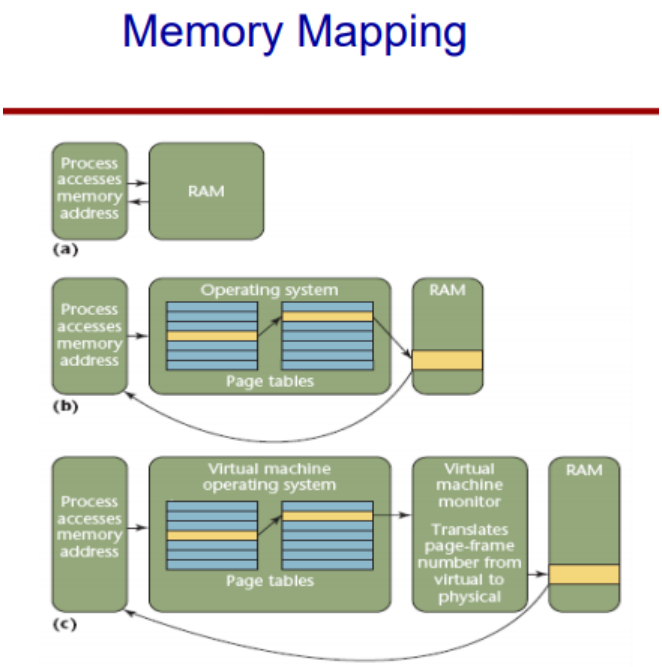
Three conditions must be met to make a chain of hashes trustworthy: (1) the first code running and extending PCRs after a platform reset *(SRTM)* is trustworthy and can't be replaced. (2) PCRs aren't resetable, without passing control to trusted code. (3) chain must be contiguos: there's no code inbetween that is executed but not hashed.

Note: TPM only works for non-virtualized environments. If virtualization is used *(clouds)*, the TPM needs to be virtualized too.

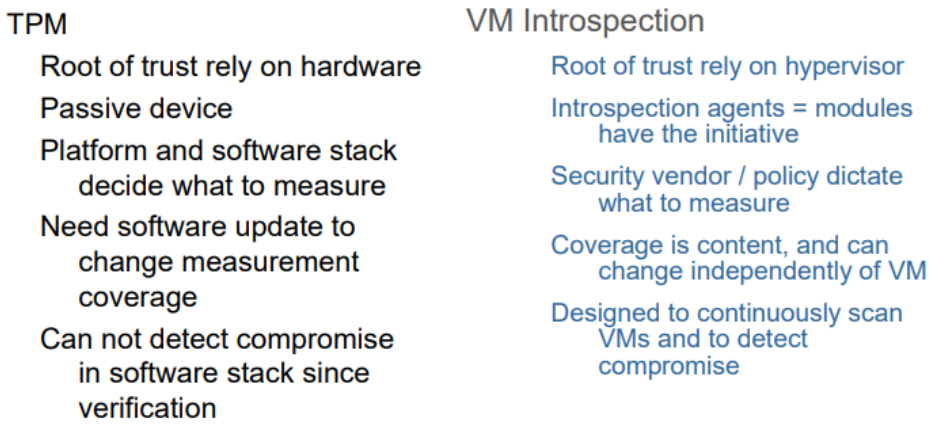


VM introspection

VMI formally defines techniques and tools to monitor the VM runtime behavior to protect the VM from internal/external attacks. Inspecting happens from the outisde to assess what's happening on the inside!



Memory mapping: requests result in direct access to the memory address. The OS layer has an active role in providing memory location access: the VMM provides an abstraction layer between each VM OS's memory management and the underlying physical hardware. VMM translates the VM-requested page frame number into a page frame number for the physical hardware and gives the VM access to that page.



There are alternative implementations of VMI:

- **Asynchronous:** the introspection VM evaluates some invariant that should hold independently of the actions the VM is currently executing
- **Synchronous:** the introspection VM monitors the execution of the other VM and, at some predefined moments, freezes the VM execution to evaluate the invariant on the status of the frozen VM, to then resume the execution of kill the VM. This requires the synchronization of the two VMs.

This introduces a semantic gap problem: the introspection VM access singles memory positions but the invariant is defined at a higher abstraction level!

This all holds as long as the VMM separates both the environment to be monitored *(monitored VM)* and the monitoring environment *(introspection VM)*. This is more expensive, but more robust than those implemented between two processes sharing some memory. **To minimize the control overhead**, a chain of trust is built where some components in the monitored VM implement some control and the introspection VM checks the intergrity of these components. *In any case, the control requires the formal definition of a process Self to be compared against the actual process behavior.*

Process Self: the process properties that determine its runtime behavior. It can be approximated through static analysis.

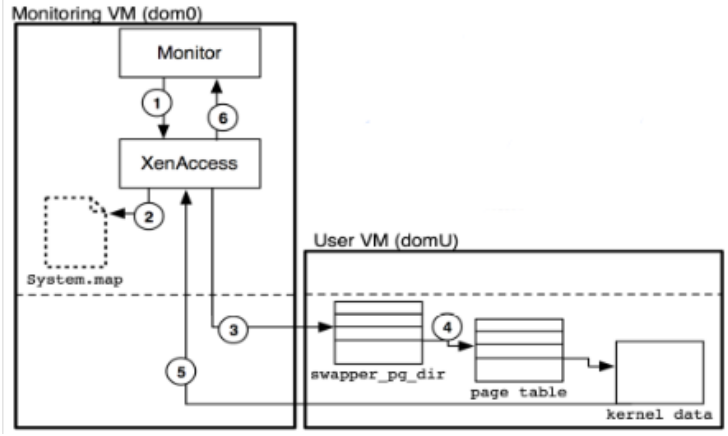
Axiom: Any difference between the process current behavior and the process self is due to an attack *(attacks against user-level processes [injecting code, diverging control-flow], attacks against the kernel [modify some kernel functionalities and the kernel behavior to hide signs of attack])*

Further advantages of VMI are its **full visibility** of the system running inside the Monitored VM and its **trasparency** *(security checks can be implemented without modifying the SW on the MonVM and they are almost invisble even to time based control)*.

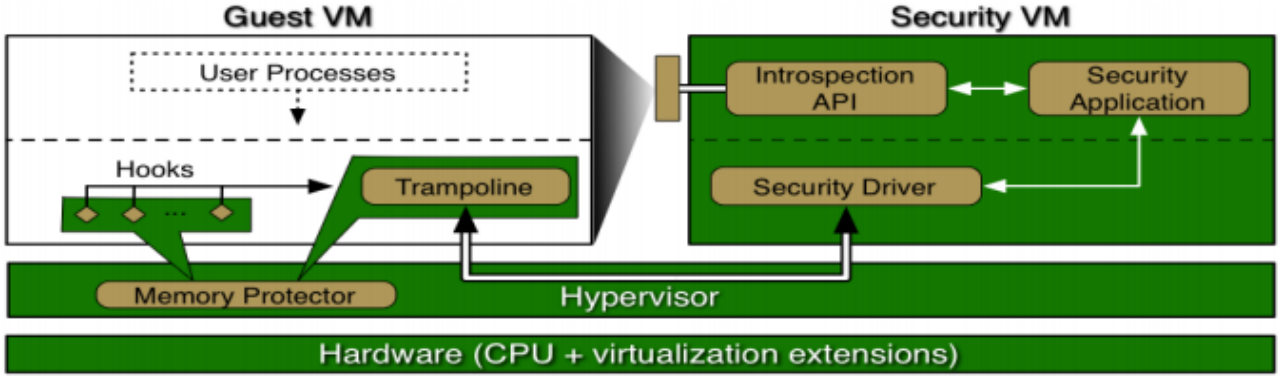
It is widely accepted that an abstract description of a process self should consider the OS calls issued by the process: any attempt to violate the security policy *(hide traces, etc)* involves some interaction with the OS. There are two alternative approaches to define the self: **Monitor and Learn** *(anomaly detection)* and **deduce the self from some representation of the process code**. The first doesn't require the source code, but it's less accurate. *From here, usual talks about default allow/deny, enriched traces, grammar generation, etc.*

To monitor memory, there are 3 possible ways:

- **Passive Monitoring:** to monitor application memory of another virtual machine we have to map the memory into an address of the monitoring one. Mapping "raw memory view" to virtual addresses and symbols requires the steps shown in the figure.



- **Active Monitoring:** monitoring application receives event notification from the Guest VM when code execution reaches one of the hooks installed in the Guest VM kernel. Hooks and all associated code are protected from tampering using hypervisor-enforced memory protections and they transfer control to the security application.



- **Locating Valuable Data:** applying formal models or obtained from supervised learning to find critical data structures within the raw memory view.

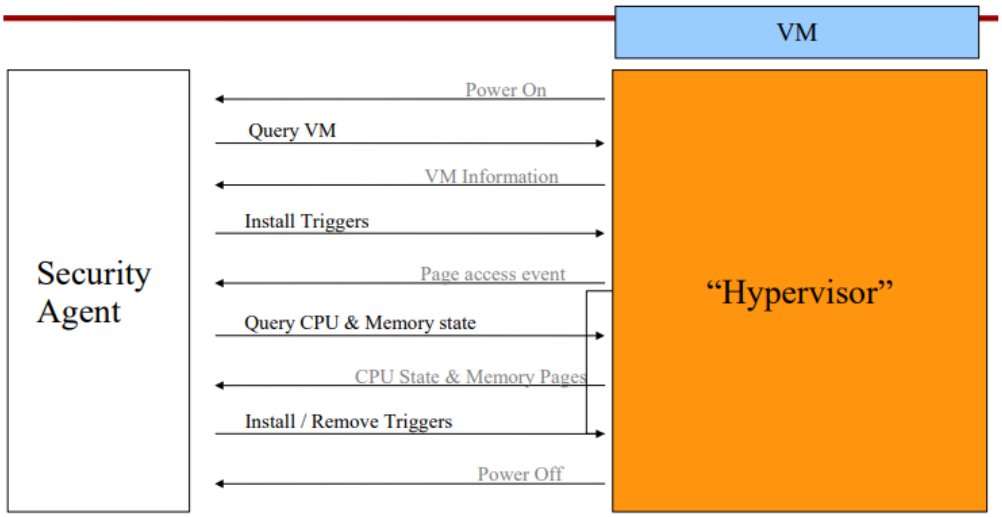
Security APIs (Vmsafe)

A new security technology for virtualized environments that can help to protect your virtual infrastructure in ways previously not possible with physical machines. VMsafe provides a unique capability for virtualized environments through an API-sharing program to develop security products.

It enables third-party security products to gain the same visibility as the hypervisor into the operation of a virtual machine to identify and eliminate malware. Security vendors can leverage VMsafe to detect and eliminate malware that is undetectable on physical machines.

Goals: better than physical *(exploit hypervisor interposition to place new security agent, and provide security coverage for the VM kernel and apps),* hypervisor as a Base of Trust *(divide responsibilities between the hypervisor and in-VM agent)*, security as an infrastructure service *("agent-less security services for VMs, flexible OS independent solutions)*

Verify-Before-Execute *(utilize memory introspection to validate all executing pages)*



Retrospective Security

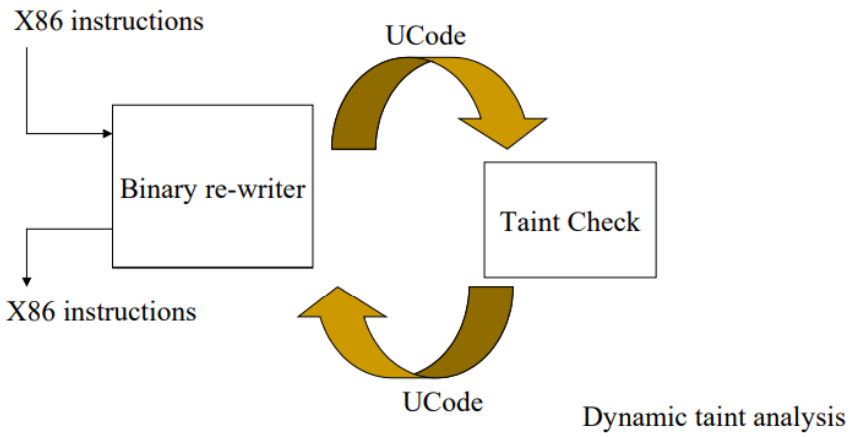
Detect whether you have been attacked in the past / you're still compromised by a past attack *(VMware Record & Replay, etc).* It's good to run more aggressive policies that aren't acceptable in production environments, or to find 0-days used to exploit your system, learning how the attacker did something, etc.

Other approaches are possible: **Threat Monitoring** *(a system can only detect and report problems, Livewire)* and **Threat Interfering** *(can actually respond to a detected threat, might terminate relevant processes and reduce resources available to the VM [starve the attacker], LycosID, muDenali)*

Tainting

Track OS-level information flow provenance by assigning a unique identifier (color) to each potential malware entry point. So it's a color-based identification of malware contaminations, resulting in a color-based reduction of log data to be analyzed.

Taint analysis should be applied anytime a malicious user input can be the vector of an attack. Very important even in the case of web applications. *Input data must be marked as "tainted"*, then monitor the program execution to track how tainted attriibutes propagate and check when it's used in dangerous ways.



Example: TaintCheck, TaintSeed, TaintAssert, Exploit Analyzer

Types of attacks detected by a TaintCheck are the **overwrite attack** *(jump targets [return addresses, function pointers, offsets], altering points to existing/injected code)* and **format string attacks** *(a malicious format string to trick the program into leaking data or into writing an attacker-chosen value to an attacker-chosen memory address.*

TaintCheck doesn't require source code or specially compiled binaries, it reliably detects most overwrite attacks *(ATPhttpd, Cfingerd, Wu-ftpd all detected)*and has no (known) false positives.

It impacts performance on the CPU bound *(RedHat 8.0 example, 37 times slower, 8s -> 305s),* on short-lived programs *(on long lived programs the penalty is acceptable, but in short-lived programs 13 times longer)* Common case: for network services the latency experienced is due to network and/or disk I/O and TaintCheck penalty isn't noticeable.

Homorphic Encryption + Enclaves

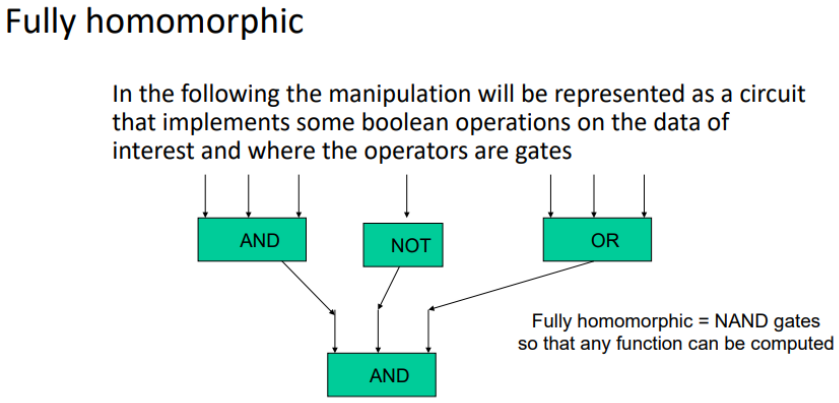
When a client stores its data on a cloud system and wants to implement some computations on it without leaking any information about data / which data is used, it requires some proper encryption scheme.

The holy gray of encryption is the **homorphic encryption**.
Let *R* and *S* be sets, *E* an encryption function *R -> S*, *E* is:

additively homomorphic if	*E(a+b) = PLUS(E(a), E(b))*
multiplicatively homomorphic if	*E(a*b) = MULT(E(a), E(b))*
mixed-multiplicatively homomorphic if	*E(xy) = Mixed-Mult(E(x), E(y))*

E is **fully homorphic** if there are no limitations on the manipulations that can be performed.

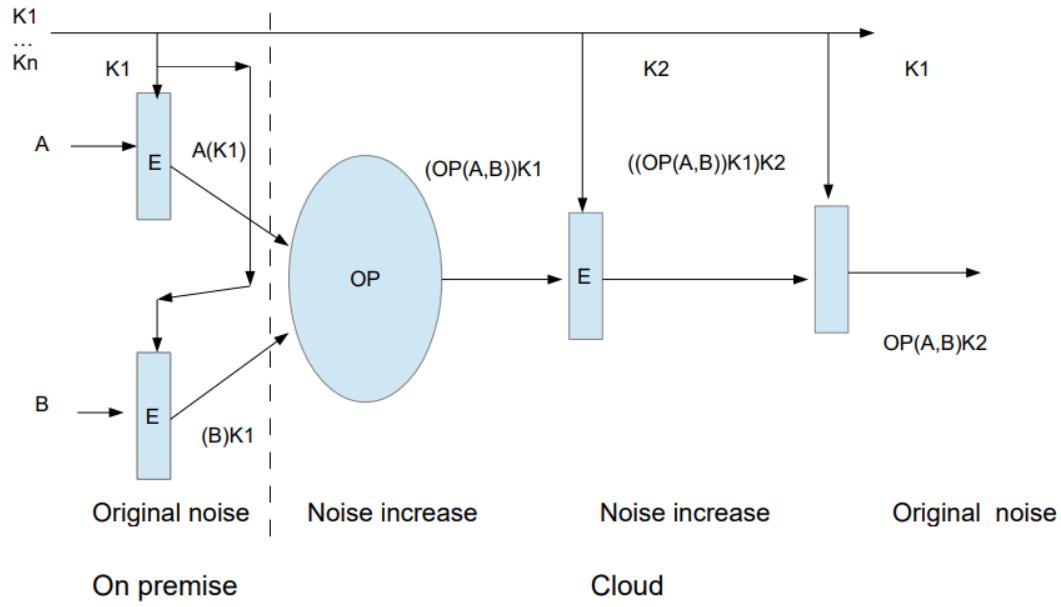
Provider stores the data and does the computation, while inputs are encrypted by the client. The output is transmitted to the client that decrypts it. There's no trivial solution that is accepted, because almost all the computation has to be executed by the provider to prevent problems.



This means that any computation can be expressed as a Boolean circuit: a series of addition and multiplications. Using such a scheme, any circuit could be homomorphically evaluated, effectively allowing the construction of programs which may be run on the encryptions of their inputs to produce an encryption of their output. *Since such a program never decrypts its inputs, it could be run by an untrusted party without revealing its inputs and internal state.*

But this introduces further **constraints**:

- **No optimization of the computation is possible:** minimization may not be applied due to information leaks. If leaks are acceptable though…
- **Size of the output must be fixed in advance:** this means that the number of output wires in the circuit must be fixed in advance. If I request all of my files that contain a combination of keywords, I should also specify how much data I want to retrieve, because from my request the cloud will generate a circuit for a function that outputs that much size of the correct files. Output will then be either truncated or padded to prevent leaking something a priori.
- **Semantic security against CPA:** (chosen-plaintext attacks), given a ciphertext *c* that encrypts either *m0* or *m1*, it's hard for an adversary *A* to decide which of the two values *c* encrypts, even if it's allowed to choose both of them.
- **Encryption scheme deterministic -> can't be semantically secure:** it means only one ciphertext that encrypts a given message. An attacker can easily tell whether *c* encrypts *m0* by encrypting *m0* and by checking if the result is *c*.
- **So a semantically secure encryption scheme must be probabilistic:** several ciphertexts encrypt the same message, and the encryption chooses one randomly *(according to some distribution)*.



To evaluate a function f in e':

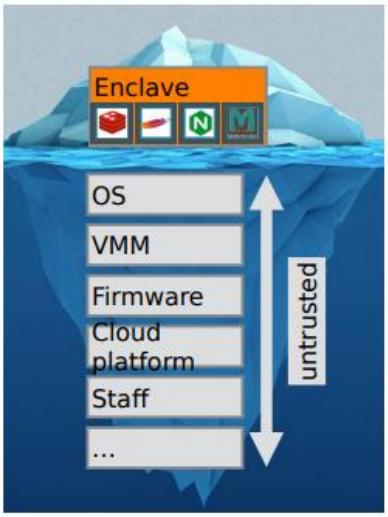
1. **we express f as a circuit, topologically arrange its gates into levels**
2. **scan sequentially the levels and for a gate at level *i + 1*** (e.g., an Add gate)
    a. take as input the encrypted secret key sk_i and a couple of ciphertexts associated to output wires at level *i* that are under pk_i
    b. homomorphically evaluate *D*Add to get a ciphertext under pk_i+1 associated to a wire at level *i + 1*.
3. **output the ciphertext associated to the output wire of *f***. Putting the encrypted secret key bits sk_1, …, sk_a in the public key of *e'* is not a problem for security because these bits are indistinguishable from encryptions of 0 as long as *e* is semantically secure. Last step: reduce the complexity of the key, instead of several public keys we have the same key for all the level (no information is leaked by revealing the encyption of a secret key under a public key, circular security)

But is it practical? It seems that, in the case of a Google search, performing the process with encrypted keywords would multiply the necessary computing time by around 1 trillion *(10^{12} – Gentry estimation)*. By exploiting Moore's law, after 40 years an homomorphic search would be as efficient as a search today.

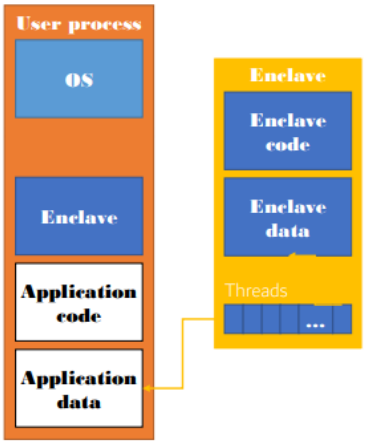
This implies an HW solution: **Trustworthy data processing in untrusted clouds.** The trust issue is from the:

I. **Provider perspective**: the Cloud provider doesn't trust its users, and so uses VMs to isolate them from each other and from itself. VMs only provide a one way protection!
II. **User perspective:** users trust their apps and must implicitly trust cloud provider. Apps implicitly assume trusted OS or system adming as well!

Intel SGX. Users create HW-enforced trusted environment *(enclave, an isolated memory region)*. It supports unprivileged user code, while protecting against strong attacks. It also allows remote attestation, and it's easily available on commodity CPUs.

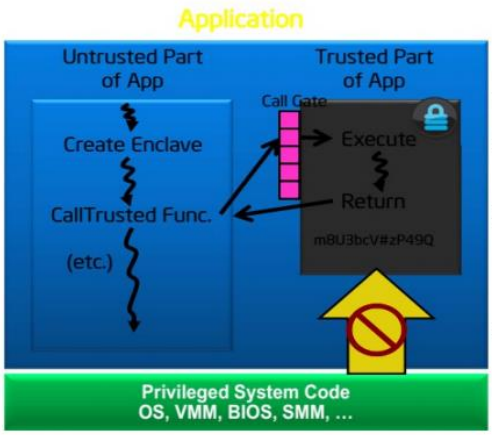


This way we are able to create a **Trusted Execution Environment (TEE)**: you're able to have your own code and data, to control entry points, provide **C** and **I**, have full access to application memory, support multiple threads, etc.



Intel SGX is an extension of the Instruction Set Architecture (ISA) in Intel CPUs from 2015.

This solves the fact that **apps aren't protected from privileged code** attacks, that may be targeted before and exploited to gain control of that "privilege". It also helps in **reducing the attack surface**, reducing it to just the App itself and the Processor: subverting OS/VMM or Bios or even Drivers can't steal app secrets!

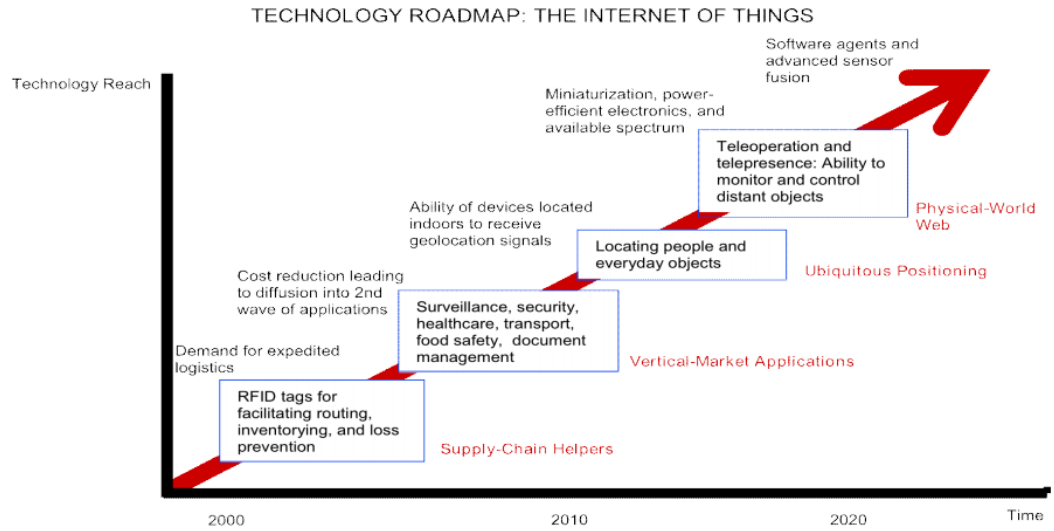


1. App is built with trusted and untrusted parts
2. App runs & creates enclave which is placed in trusted memory
3. Trusted function is called; code running inside enclave sees data in clear; external access to data is denied
4. Function returns; enclave data remains in trusted memory

SGX-LKL runs unmodified Linux apps in SGX enclaves.

Security of Internet of Things

IoT = Extending the current Internet and providing connection, communication, and inter-networking between devices and physical objects. It goes beyond the traditional Machine to Machine and covers a variety of protocols, domains and applications *(medical sensors, physical activity tracking, home heating, etc)*. Its diffusion is increasing very rapidly in recent years.



All this data that is being collected, can't answer all the questions that may arise thinking about it: *which data is collected, who is responsible for that data, etc.* Once data leaves your system, many problems that we already discussed may arise.

Example #1: 2014 a smart fridge sending spam

Example #2: If your smart lights are off, your thermostat has turned down your central heating and all your doors and windows are locked its pretty obvious you aren't at home and the history of your thermostat setting shows when you are likely to return.

Any smart device increases the system's attack surface *(attacks **to** the device)* and may store some malware to attack your system *(attacks **from** the device)*. In the "past", most were of the first type, but second type is increasing a lot due to "Internet 4.0" that includes a lot of devices with code that can't be easily accessed and tested.

A simple enough OS that is chosen by manufacturers is Busybox, a stripped down version of Unix. Some devices need a fair amount of bandwidth and may make a direct 802.11 wifi connection to the router, or some others are meant to work in a group and use wireless protocols like ZigBee, etc.

Some devices *(like CCTV security camers)* connect directly to the internet and have dedicated IP address, so are directly accessible over the internet, bypassing the need for a cloud service provider or gateway. Many IoT devices are exposed directly to the internet by enabling port-forwarding.

IoT security is an afterthought: it's already hard to create a cheap, reliable, resource-consttrained device that can connect to a wireless network while also using little power. For this reason, many attack vectors:

- **Weak Password:** to simplify the login, manufacturer offers default login combinations
- **Lack of Encryption:** many IoT devices don't support encryption
- **Backdoor:** manufacturers put "hidden" access mechanisms to simplify the support. But once the backdoor is known, instead of removing it, they just try to make the access harder
- **Internet Exposure:** since devices accept internet traffic and have little/no security, attacks.

For this reason, it's a good practice to always change the default password. To remove devices with telnet backdoors, you need to discover these through IoT scanners with an IoT search engine *(Shodan)* to reveal if your devices are vulnerable based on the IP address of the scanning computer. Then, when the device asks to open the firewall to expose it to the internet, the answer is almost always no. Finally, port scanning all the machines should be enough to prevent these usual problems.

A first classification

Via Performance:

- **Ultra-constrained node:** energy harvesting limits radio trasmissions to conserve power.
- **Constrained node:** most likely running on a battery and SW optimized for that. Still minimizing radio transmissions.
- **Mainstream node:** more complex interactions with the context since there's room for more local operation, instead of sending all data upstream.
- **Gateway node:** advanced software and runs from main power. Multiple radios to support local net.

Via Classification:

- **Simple node:** not aware of the rest of the local network. It collects and reports information to the specified destination. *It may be the first three of the previous classification.*
- **Smart node:** fully aware of all other network nodes mainly via SW, local topologies and authorized interactions between nodes in the same network.
- **Access node:** edge box to connect the local network to the Internet via whatever broadband link is appropriate for the application.

Note on node's architecture: multiple processors may be required to reduce power consumption since the only right amount of processing power is activated at any moment to handle the task.

Most real time embedded designs use a flash memory to store the program, a SRAM to store code and data, and finally a ROM to hold the basic system description.

JTAG is a common hardware interface that provides your computer with a way to communicate directly with the chips on a board. The JTAG interface gives a way to test physical connections between pins on a chip. When using JTAG to debug a chip, we are making sure pin Ap on chip A is physically connected to pin Bp on chip B, and that all those pins are functioning correctly. Since JTAG gives you direct hardware access to a device, it is a tool for security research.

We spoke about JTAG because through its interface (+ OpenOCD) it supports "Hardware-based Software Debugging" that establishes that an attacker with JTAG access *(unrestricted physical access to the device)* can read/write on memory, pause execution of firmware *(set breakpoints, watchpoints)*, patch instructions or data into memory, inject instructions directly into pipeline of the target chip *(without modifying memory!)*, etc. **Fun fact:** JTAG works through pins on a chip and manufacturers know about what we just said, so they intentionally fuse these wirings to effectively reduce the damage *(which can be repaired anyway by a very talented guy eheh)*.

Some attacks

Hack Attack: a SW attack. Examples include viruses and malware which are downloaded to the device.

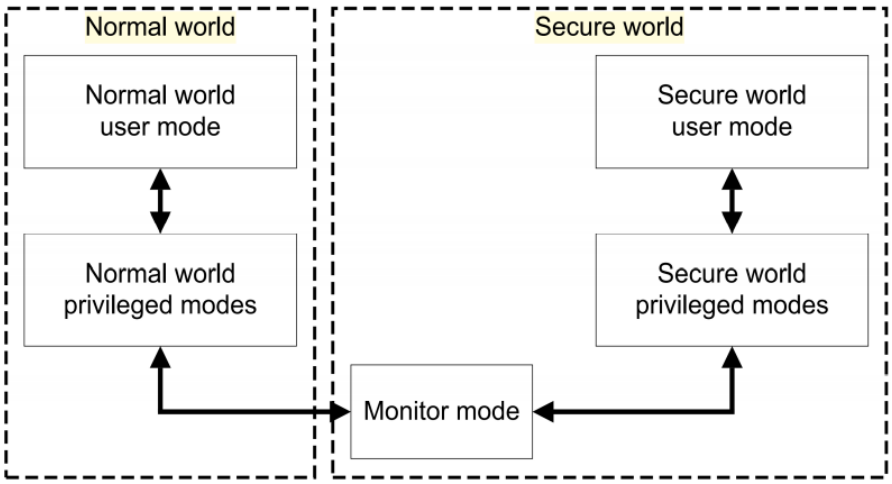
Shack Attack: A low-budget HW attack, using equipment bought on the high street from a store. The attacker has to have physical access to the device, and then it can attempt to connect using JTAG debug and built-in self test facilities. He will passively monitor the system and perform simple active HW attacks, like forcing pins and bus lines to be at high or low voltage, etc.

Lab Attack: attacker with access to laboratory equipment *(e.g. electron microscopes)* can perform unlimited reverse engineering at transistor-level detail for any sensitive part of the design, including logic and memories. Here attackers can reverse engineer a design, attach microscopic logic probes to silicon metal layers, flitch a running circuit using lasers/etc, monitor analog signals *(for cryptographic key analysis)*, etc.

Class attack: an easily reproducible attack that can be used to break a whole generation *(class)* of devices. The most widely published attacks are the ones like breaking SW restrictions on games consoles and the content protection on DVD movies. This attack shifts the balance of the economic argument in the favour of the attacker if the number of attackable devices is sufficiently high.

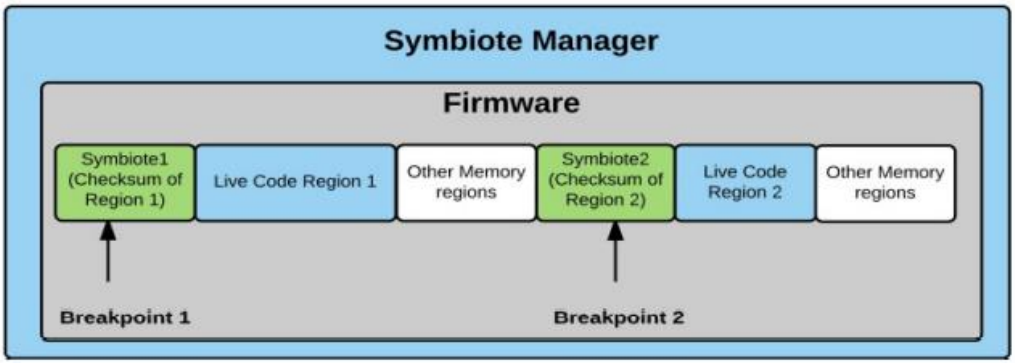
TrustZone: a hardware architecture that allows to construct a programmable environment to protect from attacks. A partition of all HW and SW resources in either the Secure World *(for the security subsystem)* and for the Normal World *(anything else)*, both run on a single physical core *(giving two virtual ones, one Non-Secure and one Secure and the mechanism to switch context "Monitor Mode")* in a time-sliced fashion. It is similar to the Intel Enclave.

NS-Masters will have Non-Secure (NS) bits set as "high" in the HW, so that NS masters can't access Secure-Slaves.



The entry to Monitor is triggered by a dedicated instruction, the *Secure Monitor Call (SMC)*. The Secure world has a little more flexibility, because it can directly write on the CPSR *(Current Program Status Register)*.
For debug, we use two bits: SUIDEN (invasive) and SUNIDEN (non-invasive) in a Secure privileged access only CP15 register.

Doppelganger: a host-based intrustion detection solution for embedded devices. It can detect both kernel and application lvl atks in embedded devices. It analyzes Live code regions *(executable parts of the firmware)* and randomly inserts its watchpoints into them: they contain the CRC32 checksum of the randomly selected regions.

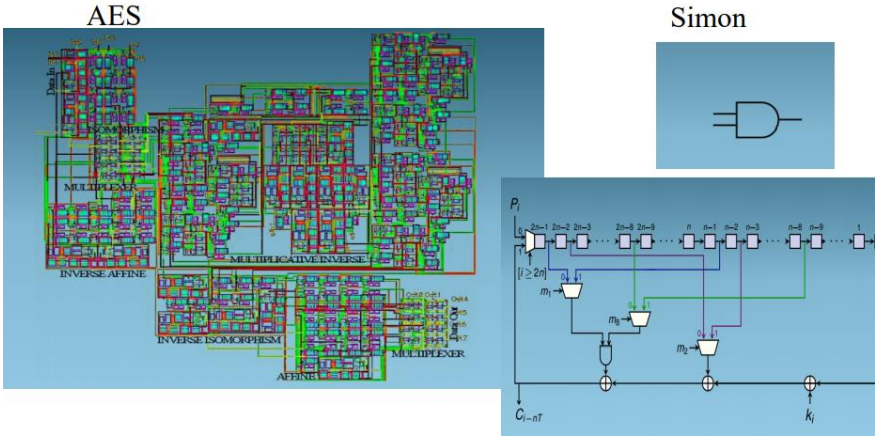


For each "symbiote" in memory, it stops the execution process, it compares the current memory area checksum of the symbiote one and searches for mismatches.
Note: doesn't defend against atks that load code in the dynamic memory.

Returning to IoT……
For the update and patch thing, encryption keys may need to be stored on the device itself: problem of the trust on the manufacturer of the device.

- **SIMON & SPECK encryption for IOT**

Many algorithms are highly optimized for particular platforms / use cases, while S&S is highly adaptable *(koala and crow image..)*. They're block ciphers families *(each with ten algs)*, they support block sizes of 32->128 bits



Anyway, IoT devices can also imply physical attacks like voice commands inaudible to humans but on a frequency that can be received by the device. *(DolphinAttack)*

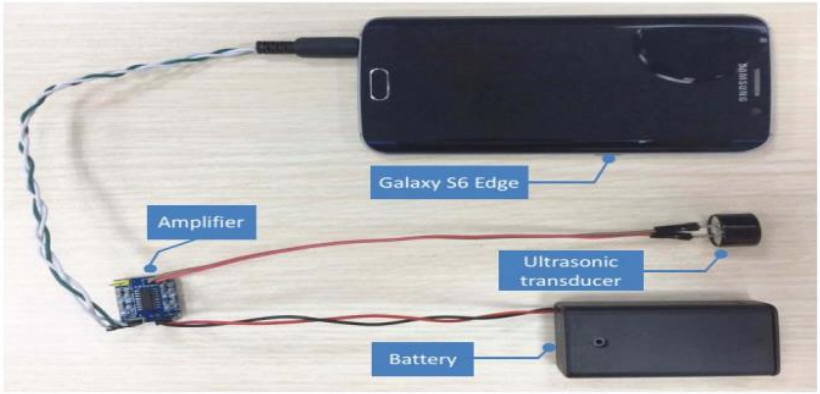


Figure 11: Portable attack implementation with a Samsung Galaxy S6 Edge smartphone, an ultrasonic transducer and a low-cost amplifier. The total price for the amplifier, the ultrasonic transducer plus the battery is less than $3.

New taxonomy on attacks can be clustered into 4 broad types of attacking behavior:

1. **Ignoring the functionality**
   The attacker ignores the intended physical functionality of the IoT device, and considers it only as a standard computing device. It may, for example, penetrate the victim's home network and infect his computers by exploiting IoT devices. These are a serious threat but the least interesting ones because they're applicable to any networked device.
2. **Reducing the functionality**
   The attacker tries to kill or limit the designed functionality of the IoT device *(lights not turning on, etc)*. This can be done to be annoying or to inflict financial loss, etc. In particular, the attacker can use ransomware to temporarily lock an expensive physsical device and demand a large payment to restore its functionality.
3. **Misusing the functionality**
   These attacks use the physical device, but in an incorrect/unauthorized way. These attacks are likely irritating pranks rather than serious problems *(turn on all lights when you go on a long vacation – cit.)*
4. **Extending the functionality**
   Extends the functionality of the IoT device to achieve a different/unexpected physical effect. *(Roomba can unlock the front door – cit.)*.
   Example: covert channel on LED lights. Entering in a system is "easy", extracting data in a reliable and constant way is the hard part. So the basic idea is to misuse the LED's API to switch back and forth between two light intensities very close for the human eye (impossible oto distinguish) but robustly distinguishable by a light sensor.

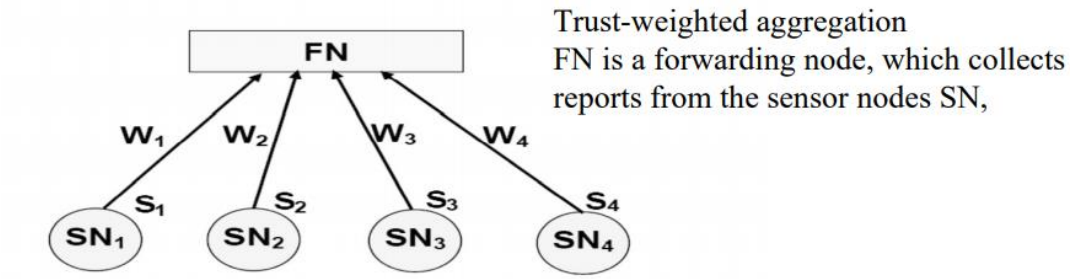
Anomaly Detection

A countermeasure to detect fake data injection into an IoT systems.
We consider correlations across three different domains:

- **Temporal correlation:** is the dependency of a sensor's reading on its previous readings. It models the coherence in time of the sensed physical process.
- **Spatial correlation**: is the dependency in readings from different sensors at the same time. It models the similarities in how the sensed phenomenon is perceived by different sensors.
- **Attribute correlation:** is the dependency in readings that are related to different physical processes. It models physical dependencies among heterogeneous physical quantities such as temperature and relative humidity.

Usually a combination of these different kinds is used.

- *Outliers and outlier detection (Trust Based systems, with trustworthiness [0,1])*



OWASP https://owasp.org/www-project-top-ten/

- **Insecure Web Interface**

How Do I Make My Web Interface Secure?

A secure web interface requires:

1. Default passwords and ideally default usernames to be changed during initial setup
2. Ensuring password recovery mechanisms are robust and do not supply an attacker with information indicating a valid account
3. Ensuring web interface is not susceptible to XSS, SQLi or CSRF
4. Ensuring credentials are not exposed in internal or external network traffic
5. Ensuring weak passwords are not allowed
6. Ensuring account lockout after 3 -5 failed login attempts

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Insufficient Authentication/Authorization**

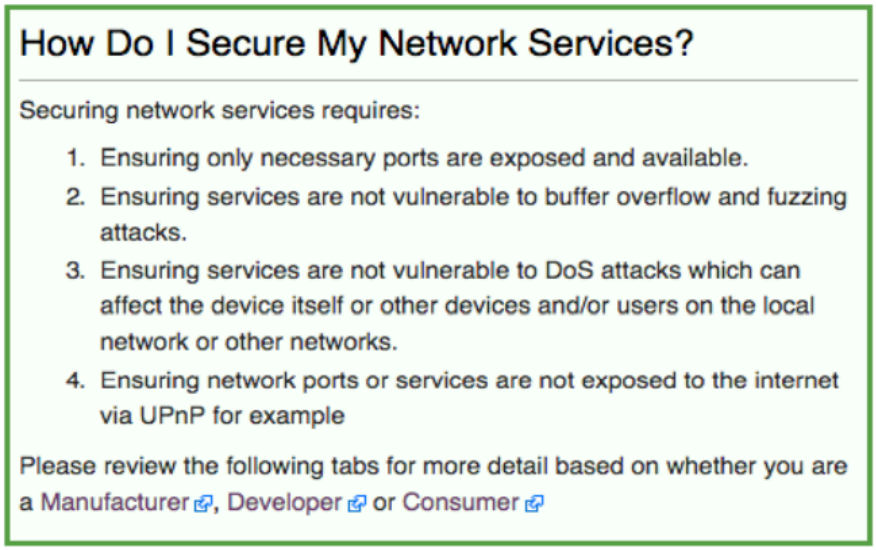
How Do I Make My Authentication/Authorization Better?

Sufficient authentication/authorization requires:

1. Ensuring that the strong passwords are required
2. Ensuring granular access control is in place when necessary
3. Ensuring credentials are properly protected
4. Implement two factor authentication where possible
5. Ensuring that password recovery mechanisms are secure
6. Ensuring re-authentication is required for sensitive features
7. Ensuring options are available for configuring password controls

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Insecure Network Services**



How Do I Secure My Network Services?

Securing network services requires:

1. Ensuring only necessary ports are exposed and available.
2. Ensuring services are not vulnerable to buffer overflow and fuzzing attacks.
3. Ensuring services are not vulnerable to DoS attacks which can affect the device itself or other devices and/or users on the local network or other networks.
4. Ensuring network ports or services are not exposed to the internet via UPnP for example

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Lack of Transport Encryption**

How Do I Use Transport Encryption?

Sufficient transport encryption requires:

1. Ensuring data is encrypted using protocols such as SSL and TLS while transiting networks.
2. Ensuring other industry standard encryption techniques are utilized to protect data during transport if SSL or TLS are not available.
3. Ensuring only accepted encryption standards are used and avoid using proprietary encryption protocols

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Privacy Concerns**

How Do I Prevent Privacy Concerns?

Minimizing privacy concerns requires:

1. Ensuring only data critical to the functionality of the device is collected
2. Ensuring any data collected is properly protected with encryption
3. Ensuring the device and all of its components properly protect personal information
4. Ensuring only authorized individuals have access to collected personal information

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Insecure Cloud Interface**

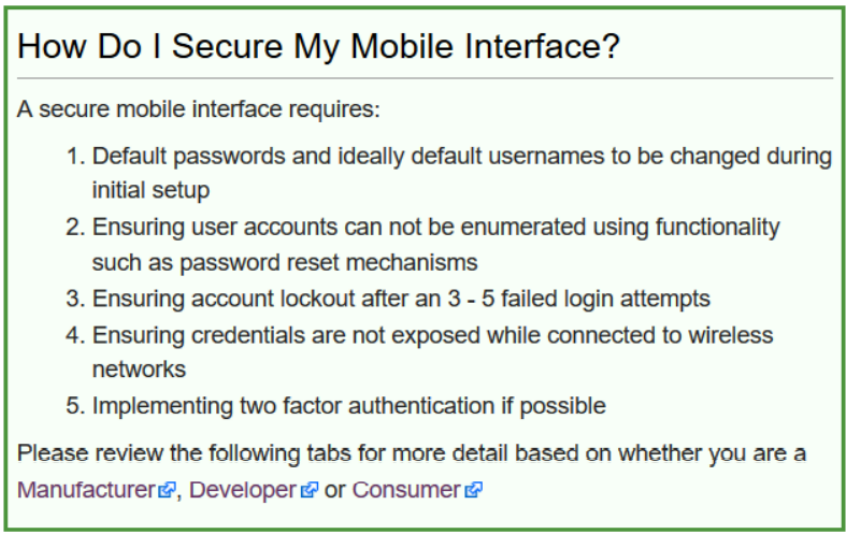
How Do I Secure My Cloud Interface?

A secure cloud interface requires:

1. Default passwords and ideally default usernames to be changed during initial setup
2. Ensuring user accounts can not be enumerated using functionality such as password reset mechanisms
3. Ensuring account lockout after 3- 5 failed login attempts
4. Ensuring the cloud-based web interface is not susceptible to XSS, SQLi or CSRF
5. Ensuring credentials are not exposed over the internet
6. Implement two factor authentication if possible

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Insecure Mobile Interface**



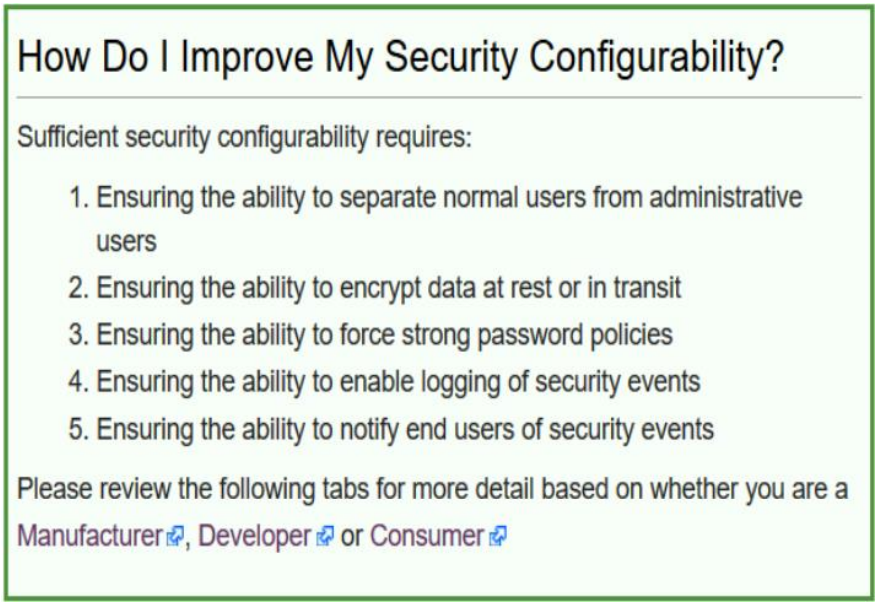
How Do I Secure My Mobile Interface?

A secure mobile interface requires:

1. Default passwords and ideally default usernames to be changed during initial setup
2. Ensuring user accounts can not be enumerated using functionality such as password reset mechanisms
3. Ensuring account lockout after an 3 - 5 failed login attempts
4. Ensuring credentials are not exposed while connected to wireless networks
5. Implementing two factor authentication if possible

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Insufficient Security Configurability**



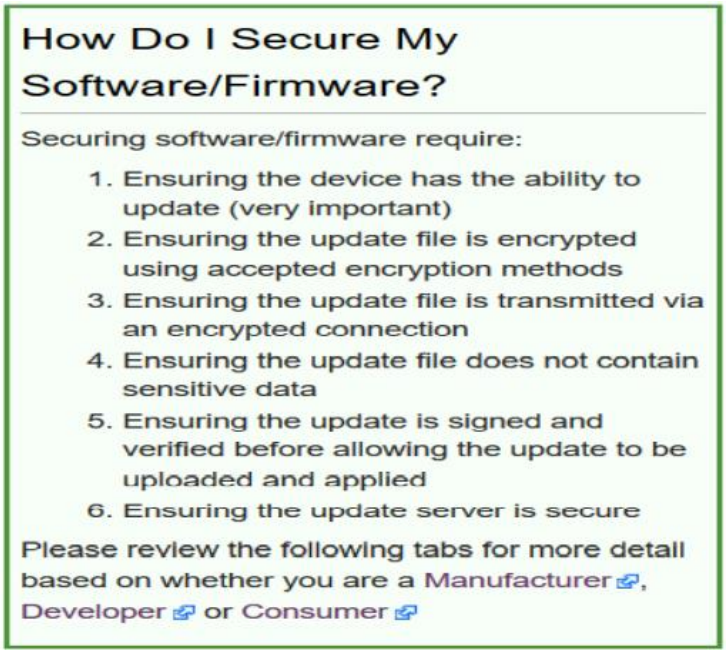
How Do I Improve My Security Configurability?

Sufficient security configurability requires:

1. Ensuring the ability to separate normal users from administrative users
2. Ensuring the ability to encrypt data at rest or in transit
3. Ensuring the ability to force strong password policies
4. Ensuring the ability to enable logging of security events
5. Ensuring the ability to notify end users of security events

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Insecure Software/Firmware**



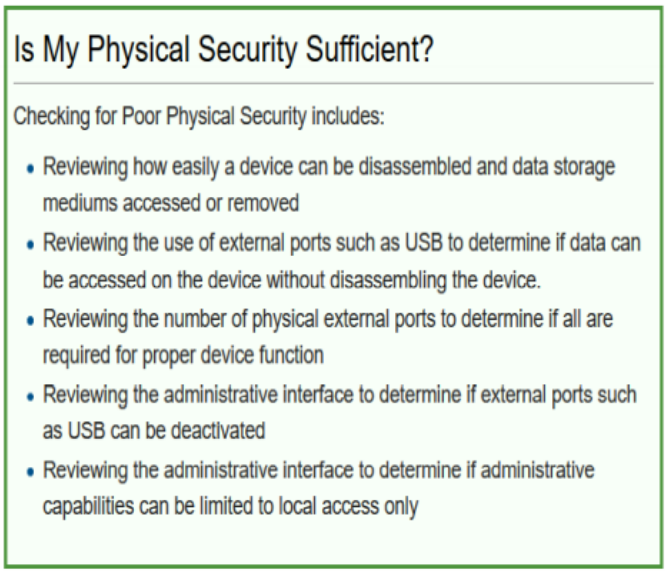
How Do I Secure My Software/Firmware?

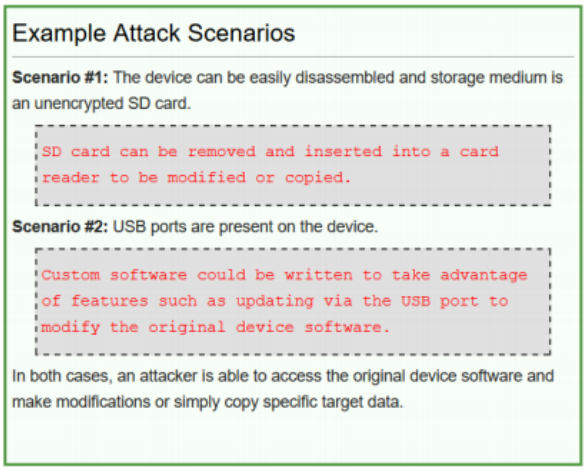
Securing software/firmware require:

1. Ensuring the device has the ability to update (very important)
2. Ensuring the update file is encrypted using accepted encryption methods
3. Ensuring the update file is transmitted via an encrypted connection
4. Ensuring the update file does not contain sensitive data
5. Ensuring the update is signed and verified before allowing the update to be uploaded and applied
6. Ensuring the update server is secure

Please review the following tabs for more detail based on whether you are a Manufacturer, Developer or Consumer

- **Poor Physical Security**





OWASP IoT Framework Assessment

A useful benchmark for vendors to produce more robust IoT development frameworks to address the IoT security issues. Evaluation criteria are broken down into four sections:

1. **Edge**
   The physical device that makes up the IoT ecosystem. It's usually heterogeneous, so an ideal framework will provide cross platform components so that edge code can be deployed anywhere from bare metal, to an embedded OS, to a mobile OS, etc. Encrypted communications should be e2e wherever possible. Data should be stored in a local storage, and as usual a robust logging is needed.
2. **Gateway**
   It will often support weak edge devices, or allow edge devices a bridge networks to cloud components. It offers an easy interface between an insecure (but trusted) local network and a secure connection to the untrusted public internet. It has a security critical role. Again, strong authentication, logging, anomaly detection, etc.
3. **Cloud Platform**
   It typically consist of a data storage layer (*e.g. DB*), a web interface, etc. The access is typically restricted, especially to the supporting infrastructure. Encrypted communications, secure web interface, authentication (multi factor), encrypted storage, etc.
4. **Mobile**
   Special care should be paid to this, because they may be deployed beyong the boundaries of device management and still be granted privileged accesses while still falling easily into malicious hands. The authentication requirements must be at least equal or greater than to other components. The stored data should be limited so that if it's stolen, the impact is reduced. It should be possible to revoke capabilities and there's a strong need for an audit trail.