2. Digging Deeper into Computer Vulnerabilities

Before I can discuss solutions, we need to dig much deeper into why we have computer vulnerabilities. In a later chapter, I will refer to each of these issues and how they disappear when we use different approaches and paradigms in cybersecurity.

I was reading a lot about the spyware Pegasus from the NSO-Group. But as hard as I try, I cannot see Pegasus and other similar products and even the unregulated "Cyber-Mercenary" industry as the reason for our problems.

The mainstream opinion sees this differently, but I don't see the unregulated cyberweapon industry as a cause for our problems. Sure, if we don't do anything, then it becomes a huge problem. For now, it is much better to consider their products as early warning signs. If malware is done by more maliciously minded people, Pegasus-type malware could have left in all visited systems a hidden backdoor that could be used for cyberwar or surveillance reasons.

Additionally, Pegasus was not used on more systems because it created so much information that its extraction and analysis took a lot of labor/staff for each instance in the field.

Overview

I came up with at least 12 problem or issue categories that contribute to or even cause many of our vulnerabilities and problems within cybersecurity. I am not saying that these problems have not been seen or recognized before, but they are not solved. I have not listed them in a particular order. Each of these issues is worth our attention independent of others.

In case you just want to acknowledge them and not get further in-depth into them, here they are briefly mentioned:

(1) **"Software is invisible"**. Software can only be seen/validated indirectly; it requires trust in the entire ecosystem to trust any part; this can be rephrased as: "Software must be trusted (blindly)", or I heard it too often: "someone else will (hopefully) see the problem(s) if there are some".

(2) **"Software is covertly modifiable"**. The emphasis is on "covert". Unfortunately, modification detection is unreliable, particularly if we have mutable software responsible for detection. There are too many blindspots where covert (temporary) modifications could happen.

(3) **“Every tool/component could be compromised”**. We must constantly question our trust in whatever we use. This can also be rephrased as: “Any software can be dangerous - potentially only temporary”.

(4) **“Attacker chooses methods and timing”**. Attackers have the first-mover advantage. Defenders must be prepared anytime for anything that an attacker can do.

(5) **“Attackers know more (about vulnerabilities)”**. Attackers know more about the weak spot they use; additionally, one vulnerability is sufficient, while defenders need to know (and fix) all. Defenders are often surprised by the used attack (details).

(6) **“Attackers can adapt to (known) methods of detection”**. Attackers don’t want to be detected. They could change their appearance, remove revealing data, or modify attack detection tools to remain undetected.

(7) “**Attacker can get away unidentified**”. Attackers can remove their traces and misdirect forensics. Without identifying attackers, we cannot hold them accountable or deter them.

(8) “**Secrets are unreliable in defense**”. Defenders should not build their defenses on secrets they assume attackers don’t have. In particular, secrets known by humans could also be known by attackers.

(9) “**Defenders can be compromised**”. Humans involved with the essential decision or process competencies can be blackmailed or deceived; they could be turned into traitors with enough pressure.

(10) **“Software output could be faked (late)”**. We communicate with software via its output, but we should be careful in trusting what we hear or see within the output.

(11) **“Complexity is an enemy of security”**. Security is part of a complex system; we should not trust it.

(12) **“Crypto-Keys/Units are unprotected”** Protecting (locally stored) crypto keys from being stolen or preventing misuse of crypto-devices is considered a problem, but not a critical core problem in cryptography.

All these issues contribute to circumstances that favor the attacker. Unfortunately, cybersecurity or cryptography did not contribute enough solutions to the above problems. They seem to have given up on providing help for these issues.

If you are concerned about too much techno talk on software engineering details, I was trying to mute that and keep it high-level. Still, you could jump to the next Subsection: “Cybersecurity Paradigms”, without losing the main thread.

More About …

(1) Software is Invisible

Software is an essential and pervasive part of our modern world. It powers everything from smartphones and laptops to critical infrastructure systems such as power grids, transportation networks, financial transactions, and medical equipment. Software is invisible, and we rarely think about that. This invisibility applies to both: useful, beneficial code and malware. We don't have a choice.

Furthermore, every piece of software is built on top of multiple layers of other software, developed or modified anonymously. We can only detect software via its output or effects. Misbehavior from problems is easier recognized as a bug than smoothly operating software. Intentionally inserted problems (sabotage), or worse intentional deception within software output, in one part of the system can quickly cascade onto other parts, potentially leading to significant disruptions or failures. Even small, minor modifications could have a huge impact. Due to millions of simultaneous operations, we have a severe problem in determining what might be the cause if that problem was not anticipated before. Locating problems within the software is a huge problem; it requires complex and time-intensive reasoning to narrow down the problem's location.

To ensure that invisible software operates properly and is trustworthy, we have a robust ecosystem of stakeholders responsible for ensuring software quality and reliability. This includes software developers and engineers responsible for writing and testing code. However, the best we could hope for is that software remains untouched (by attackers) and that there is a continuous turnover with newer software versions in which previous problems are fixed.

Currently, security-related and regular software features are usually mixed within the same software package. But let's assume we could separate and isolate security-related features. These security-related features require permissions, like reading, writing, or executing files, outputting data, and using or restricting services or features.

As (already) mentioned, rights and permissions are strict yes or no decisions based on separate data associated with the user or user role. The regular software does not impact this decision - it uses other software from the operating system to make these lower-layered changes/decisions. The OS is seen as neutral and incorruptible; the same applies to the data it uses. But unfortunately, the software and its data are within the system's computer memory (RAM), potentially in striking distance of attacking software.

Can an attacker modify security-relevant data or instructions? We should better ask: How can we know that attackers did not find a way to bypass file or OS restrictions?

Software operations and their data are all invisible. We could only indirectly reason or speculate that, hopefully, nothing nefarious is happening in the background. Additionally, opening software files or displaying data is not providing direct or compelling proof that what we see is a proper representation of what is (really) there. We must acknowledge that there are many layers between what is on a computer and what we can reliably detect or see.

As a thought-experiment, let's assume we could isolate security-related operations and data. Let's assume that security-related data and instructions are processed in a context where every change/manipulation to these instructions or data could become visible (as an alarm), traceable, and analyzable in the aftermath. In these circumstances, attacks are detectable and not invisible anymore. We don't need to make the entire software visible; as a start, it is probably sufficient to detect critical changes to security-relevant features or data.

Another problem with (low-level) tools is that they let us see code but don't tell us anything about its purposes. Under these circumstances, how can we be sure that there is no additional hidden (malicious) sleeper code or backdoors in the software? We have to live with some level of invisibility within our software.

(2) Software can be Modified Covertly

With many safeguards in place, there is still no guarantee that software cannot intentionally be manipulated. Do we know if a modified file was reset to its original version after the attack? Currently, malware is not too concerned about removing traces. If attacker's goal is to remain hidden, then this is not too difficult to accomplish once malware has sufficient privileges.

Additionally, modifications could happen to software or data while being loaded into RAM or the CPU's cache, i.e., just before execution on the CPU. I am fully aware that OS and CPU have some protection against these scenarios. But CPUs have multiple levels of cache and 100s/1,000s of low-level instructions. We should doubt that manufacturers and their security engineers could categorically exclude that security-related data (incl. software instructions) are protected from modifications within RAM or CPU's cache. I dare to make these statements because CPUs serve different OS systems that manage multi-tasking and multiuser features differently.

For the more technology-versed readers: CPUs do not force the OS to accept a specific way of dealing with user permissions. Access management is the sole domain of the OS, and the data structures used in dealing with permissions are included via OS algorithms and not in the CPU. The CPU ignores permissions if the OS does not make the CPU explicitly respect these permissions. The same applies to data in RAM. It is the OS that enforces memory access protection or allows shared memory. It is the OS that activates or deactivates CPU protection features. Doing a restart after some low-level changes in security is just a precautionary measure. Software/malware could fix potential problems without a restart.

This is not the place to remain too technical - but one more thing: CPUs support security rings in which higher-privileged operations get additional hardware support to protect their operation from lower-privileged data operations.

But what is being done within the rings is determined by (modifiable) software. If an attacker can change privileged software, then there is no protection. So, can this be prevented? Is this threat already being prevented? I have doubts about the last question (from what I know), but I believe it can be prevented.

There is currently no inherent part of software development and no amount of testing or verification that could prevent or protect software from hidden access. To my best knowledge, i.e., as someone who studied OS and system architectures as an outsider, there is only one concept: a hardware-based root of trust with immutable code within a separate component. I like that concept, but I'm not too fond of the immutable part. What if the code in that component is flawed? Then all machines with that component become vulnerable immediately.

Furthermore, changing the stream of bytes entering RAM/CPU with some instructions using a simple/generic software/file/data modifier is an extremely simple and versatile attack solution if this modifier software is included in the software stack.

Currently, we must (blindly) trust that no invisible background code is covertly modifying software. Could this feature be included in the kernel? Unfortunately, yes. Is it included? Probably not (yet).

It is extremely difficult to ensure software is not manipulated in some blind spot. As users, we must rely on the expertise and vigilance of dedicated professionals so that this doesn't happen. Unfortunately, their tools are always limited, and their checks apply in well-known ways advanced adversaries could foresee.

Software developers love to create versatile swiss-army knife solutions applicable to as many different situations as possible. The resulting complexity and the methods/tools used in creating code or checking their results are not designed to help in detecting covert modifications. Instead, we should assume that malicious code could be inserted unbeknown to developers. This could happen through tools used in the development or via modifications after deployment.

Without comprehensive detection methods, malicious modifications and backdoors can remain covert even within security audits. Unfortunately, it is simply too labor-intensive and expensive to distrust everything. Even if detection methods exist, we must assume they might not be good enough. We should admit that we have known or unknown blindspots in which covert modifications could happen or hide.

(3) We can't Trust any Tool

In today's world, it is important to constantly question our trust in the tools and components we use. Any tool or component could be compromised and used against us. These statements result from the complexity of technical solutions provided by humans. Still, developers could make mistakes (intentionally), leave backdoors or vulnerabilities in their code, and even sell this information as a 0-Day vulnerability for some high-dollar amount.

We rely on many auxiliary tools (editors, compilers, etc.). We usually trust them until we have evidence to distrust them. We also trust the OS correctly displays or logs software's hidden activities because we can't know any better.

It is easy to recommend being vigilant, constantly looking for vulnerabilities, and taking steps to fix them as soon as they are discovered. But that is much harder than it sounds.

It is also important to be up-to-date with the latest security measures and to stay with best practices ahead of potential attackers. Without automation and support, this could quickly become a full-time job. Constantly questioning our trust in used tools is not productive. Developers (and regular users) depend on sound help against potential attacks.

It would be a huge advantage if developers could focus on their solution and close the book on covert modifications within their tooling and scope of work.

(4)-(6) Attackers Advantage

The primary advantage of attackers is that they "(4) choose methods and timing". They can wait for the perfect opportunity to strike, whereas defenders must always be prepared for an attack. This puts defenders at a disadvantage, as they are constantly on guard and ready to respond to any potential threat.

One of the biggest challenges for defenders is that attackers are one step ahead. As soon as we deal with intentional modifications, (sophisticated) attackers "(5) know much more". Moreover, they know more about our specific weak spots and vulnerabilities. We must acknowledge that attackers only need to find and exploit one weakness (about which they know more) to access our systems or networks. On the other hand, defenders must be prepared to react to anything attackers can do - additionally, there are known vulnerabilities but no fixes yet.

Attackers need to remain hidden as long as possible. They could remove data traces and reset compromised tools to their normal state after use if modifications were required. Knowing how attackers have done attacks provides detection methods. However, attackers "(6) can adapt to (known) detection methods" quickly because these detection tools are accessible and almost impossible to keep secret. Additionally, attacker tools could stop attack-detection tools and still display information to satisfy the expectation of users who started these tools.

As a thought-experiment, if we could significantly limit attackers' what, where, and when, i.e., attacker's attack methods or the time at which it could attack, we could better prepare defenders for consequences. We would need to detect the where. This could help us mitigate the damage immediately. Also, defenders know more than attackers; they would not have an advantage but could likely be caught. Well, this is just a thought-experiment. However, if attackers do not know how they are being detected or cannot adapt to the detection method, defenders' chances of getting them caught are much better.

(7) Attacker can Get Away Unidentified

Unfortunately, attackers have a rich toolbox of advanced techniques to conceal their identity from which they can pick, and defenders have little in their arsenal. Attackers can use proxy-server, compromised devices, or false identities acquired in networks to anonymize their attack. Additionally, they can manipulate or destroy evidence of the attack. Leaving false traces or using other misdirection tactics could make it more difficult to extract evidence that could be used in attacker identification by digital forensics.

Without identifying the attacker, it is impossible to know the full scope of the attack, the methods used, and the potential data that could have been stolen. It is also harder to hold attackers accountable for their actions and take legal action against them. When attackers remain unidentified, they will likely continue to launch new attacks. They could make a business by stealing, harming, and blackmailing others.

Even advanced security measures, like intrusion detection and prevention systems, network monitoring, and incident response plans, are useless in identifying and tracking down attackers. However, we need tools that make it more difficult for attackers to remain anonymous.

(8) Secrets are Unreliable in Defense

Defenders should not build their defenses on secrets they assume attackers don't have. In particular, human secrets could also be quickly known by attackers. Currently, humans want to be put in the loop, although there is sometimes no reason to share secrets with human defenders. An example is a public key in the Public-Private Key-Infrastructure (PKI). This public key is being provided via cleartext certificates to the world so that other key-signing authorities can digitally sign and authenticate this key. There is no technical reason humans must see the key except the protocol implementation demands it. However, protecting public keys from being seen in cleartext would require a different approach to cryptography, which is discussed in issue (12) below.

Notifications among defenders announce (indirectly) that exploited vulnerabilities, backdoors, or malware were discovered and neutralized. When this happens, opportunities are lost to trap attackers. If this information could be kept secret from humans, then attackers would learn from it after they get caught more often.

Secret honey-pots or tripwires could give defenders an advantage, but these techniques (when used at scale) are standardized and could be probed using decoys or simulations. They could be detected indirectly via suspicious or revealing signals/data.

With advanced attackers, defenders should better assume that they know everything the (human) defenders could know: "Secrets are unreliable in defense". In particular, every secret only theoretically known by human defenders could be known by attackers via carelessness or traitors. As a result, it is important for defenders not to rely on exposable secrets when building their defenses.

The only valuable lesson is: keeping secrets is very unreliable. Attackers constantly find new ways to access information and exploit vulnerabilities, and defenders can never be certain that their secrets are truly safe from discovery. Even source code of security-relevant features should be published because sophisticated attackers will decompile applications and analyze the source code despite additional measures of code transformations via obfuscation.

The only reliable secret is a secret that no human can see or receive.

(9) Defenders can be Compromised

Defenders can be compromised in various ways, including phishing, social engineering, duplicated physical access keys, bribing, intimidation, and blackmail.

Also, threats to insiders and compromised third-party vendors could compromise sensitive information, disrupting operations and causing substantial loss or damage. Although systems or organizations could protect themselves against some of these types of threats with robust security measures and regular training, there is little that can be done as an effective countermeasure except for removing humans from these positions. If automated security processes can be trusted because modifications can't happen covertly or traceless, then security automation is superior to having humans in any capacity involved.

If there are situations where humans must get involved because the algorithms are creating harm or damage, it is better to have a kill switch that could be activated by many or even all present. Micromanaging security is wrong anyway.

(10) Software Output could be Faked (Late)

When communicating with software via its output, we must be careful not to trust the output. The attacker knows what we want to hear/see. We often have a confirmation bias, and we let our guard down when we see what we expect to see. This is a difficult problem emerging from compromised tools/OS. It cannot be solved easily if we cannot prevent software modifications.

What software does or detects internally and what it shows to the outside are not necessarily the same. Advanced attackers could play us with our expectations. It could redirect screen output into a temporary file while another process creates the output we want to see.

Furthermore, output is based on data that could already be manipulated with malware outside reliable tools or OS features. This problem is also related to deepfakes in audio/video publications and later in communication.

(11) Complexity and Security

New technologies are provided for products claiming additional security, and we are convinced via marketing messages that these claims are true. I am often amazed at how easy this can go. Sure, security experts are involved, and they agree with the mindset that nothing is truly safe. That we need to compromise on the reliability of security implementations is generally accepted. Still, if an academic or freelance attacker shows a breach or vulnerability, the lessons are accepted, and improvements are provided soon after. It's safe until we have the next breach. If we had this attitude toward an advanced hacker who doesn't tell us what it can do, we would already be doomed.

My biggest problem with cybersecurity is how they deal with complexity. The demand for security is already very complex. We have different user rights; users belong to different groups with different associated rights or permissions. We also have different assets, like files, network cards, video cams, microphones, and even GPUs, that need protection against attackers. And then, there are removable storage components and devices connectable via USB or other interfaces. So the situation is already very complex. If the technologies built on top of the requirement are also complex, I start doubting its security deeply.

CPU manufacturers introduce the Trusted Execution Environment (TEE), TPM (Trusted Platform Module), or T2 as a hardware root of trust. The underlying problem remains: "Complexity is an enemy of security". Whenever systems get too complex, we must assume that their security implementation is likely flawed. It is not that I don't like TEE, TPM, or T2; I don't trust them without having certain structural issues not being solved with clear changes within the underlying cybersecurity paradigms.

The CPU consists of billions of transistors, thousands of processor instructions, and microcodes used to fix systemic hardware flaws, an incredible engineering achievement. These features come with a price: complexity. Additionally, the optimization of calculations was prioritized over simplicity and security. Every feature, including security, is designed based on a highly configurable swiss-army knife concept. Attackers can even hide their malware in hardware. That inconspicuous functions are turned into malware using microcode didn't happen yet, but it is conceivable that an advanced hacker using AI could accomplish that. With modified CPUs/microcode, even uncompromised software could be turned against its user - at least, that is the threat.

However, if we imagine we could put security operations out of reach of potentially manipulating instructions from a complex system, security would be an independent, separate service to the complex system. If there is an additional physical separation and the software used within security units is independently watched for suspicious modifications, then we could have more reliable security despite a high level of complexity in other parts of the technology.

(12) Limitations in Cryptography

Encryption encodes data so that someone with the correct decryption key can access protected data. Intercepted encrypted data are therefore protected from being read or modified as long as the attacker does not have the crypto keys or the crypto device that contain the secret keys.

Unfortunately, crypto-keys and crypto-devices are usually unprotected on most systems; they can be read or used covertly. Keys are stored in the filesystem and, best-case, protected with a password. Still, if advanced malware is on the computer, there is a chance that the password is already known or the keys were stolen when the crypto software component was used. The result is: attackers will steal all keys when used by compromised software. Attackers will certainly not use expensive and time-consuming crypto-analysis to calculate the key. The protection level (i.e., key length) is for stealing keys entirely irrelevant.

Alternatively, keys could be stored in crypto-cards. Encryption and decryption happen within these cards and not in the CPU. Crypto keys are being protected from being stolen, which makes these crypto cards or devices prone to misuse. These cards do not need to be stolen; it is sufficient if attackers use them covertly. There is no gain in having crypto cards if we cannot guarantee they are not covertly misused.

Cryptography was once used almost exclusively for national security and the military. But now, it is commercially used in untrusted environments. Commercial encryption still has the impeccable reputation when the military used encryption under simplified conditions. We now have malware capable of stealing crypto-keys or misusing covertly local crypto-devices. Off course, we don't know if crypto-units are/were compromised or wrapped into some other software manipulating in-/output. But using military-grade cryptography means nothing in protecting secrecy or integrity when we assume that attacker code is attacking keys or crypto-units directly.

There are features in computer security that require protection (secrecy), authenticity, and integrity validation provided by digital signatures. Security is impossible to deliver if we have doubts about these basic functions.

Cybersecurity Paradigms

I was also trying to understand why cybersecurity did not solve the above 12 problems. The simplest answer I came up with is that cybersecurity was not stating problems as I did. Instead, they accept the consequences of these problems, not the problems themselves, which I consider solvable.

Furthermore, I would argue that none of the above problems are handled or answered by mainstream cybersecurity-related ideas. I would even say that cybersecurities' paradigms are almost detrimental to improving security - they are staying in the way of progress. Paradigms are like assumptions based on widely accepted and rejected ideas. It is important to discuss if cybersecurity professionals follow wrongly accepted concepts (and others wrongly ignored).

(1) "Do not trust CPU/OS - but still, use it".

Most cybersecurity pros share the concern about the insecurity of OS/CPU. Still, they want to keep the flexibility and versatility of the OS and adapt their security measures to threats more easily. There is no question most pros are aware that CPU's/OS's complexity is the main reason for computers' vulnerabilities. However, they are hesitant to accept the logical consequences. Security threats from apps could then become more difficult to deal with if convenient OS features are no longer available. There are several concepts in cybersecurity, like separated firewalls, separate crypto cards, or the Trusted Platform Module (TPM) with its fixed security features. These concepts have disadvantages: they are too static and rigid. These solutions can be made more adaptable and safe.

(2) "Blacklists of threats are sufficient".

Blacklists of threats, also known as "blocklists," are lists of known malicious software or harmful files used by cybersecurity to prevent systems from accessing or executing them. One of the main limitations is that they provide protection only against already known threats. New malware is not on the blacklist until it has been identified and added. There is a window of time during which the new threats could go undetected and cause harm to systems. In addition, it is still possible that blacklisted software is modified or reconfigured in a way that makes it difficult or impossible for the blacklist to detect it. For example, a malicious actor could alter malware's code to change its signature, making it appear to be a different, benign piece of software. Blacklists provide some level of protection, but they cannot be used in proactive security solutions. Blacklists ignore unknown codes. Anonymous (unknown) code must be rejected. If someone dares to put their (valuable/built-up) reputation behind it, then there is a reason for trusting it without analyzing it extensively.

(3) "Software Developers are not trusted partners".

For cybersecurity, developers are mistrusted as cause for vulnerabilities, although most were created accidentally. Vulnerabilities are normal, and no developer should be blamed for them. Medical doctors, lawyers, and financial advisers already have self-regulating rules providing a minimum level of quality control for the public. Anonymous developers could create or use exploits (outside their main job) without consequences. The mentioned professions would not allow questionable behavior. Dealing with untrustworthy developers is a huge problem that cannot be solved technically, i.e., by analyzing developers' work products alone. Unfortunately, developers' hard-earned reputations are not used as a tool by cybersecurity - but should be included asap.

(4) “Single-Unit-Security”.

It is common software engineering practice to bundle similar features, turned into swiss army knives, and use them in as many applications as possible. This is a generally accepted design principle in software development. Fewer but configurable components used in similar tasks are considered better solutions. Super-specialized features are only accepted for high-performance tasks. Don’t use more than one component for security. But simplicity in security is the same as performance, a legitimate reason to deviate from general design principles. Multiple (concurrently used) security units watching each other for manipulations could help us create a much safer system; however, this proposal conflicts with general design principles.

(5) “Security needs humans in the Loop”.

Humans are important because security is so complicated that humans must be directly involved in many operational aspects. However, humans can be deceived or frustrated in switching off security measures - as part of attacker’s plan. The biggest problem with humans is that they believe they should know secrets used in defense. However, humans can be compromised, and the less human involvement we have in security, the better for speed and quality of security-related reactions and decisions.

Our deviations from the above list of established cybersecurity paradigms will lead to our proposed solutions presented and discussed in a later chapter.

Institutional Resistance Against Better Security

I talked with many security experts, and some described an almost overbearing pressure by governmental organizations or institutions to keep smartphones or encryption vulnerable.

The supposition we are making the world safer by keeping smartphones and encryption vulnerable is a big mistake.

When we discuss why computers are vulnerable, we must address the issue that law enforcement and intelligence organizations think they have an overwriting, superior reason to listen to phone calls and locally stored data. Pre-Internet, the same organizations were doing their job without doing mass surveillance on billions of devices. Edwards Snowden’s revelations on NSA’s capabilities were pretty important in June 2013. Now, with the Pegasus spyware, we have malware that is more than a proof of concept. Smartphones could be turned into mass surveillance tools for governments or private/criminal entities. Smartphones could also be turned into cyberwar weapons that could tumble governments.

Pre-smartphone, governments used separate listening devices, or spycams, to get the same information. I hope this book provides overwhelming evidence that smartphones are too dangerous to be used as spyware, particularly if they could have only insufficiently protectable backdoors. Law enforcement or intelligence services should not have special privileges to collect information via smartphones.

It seems we have already opened too many doors. But if we leave or keep any vulnerabilities or backdoors open, I will describe an inevitably uncontrollable mess in the following chapters.