2028 - Hacker-AI and Cyberwar 2.0+
(Securing our Future: Proactive Resilience through Separated Security Measures)
ISBN: 9798375252568

11. Countermeasures: Technical Solutions For Hacker-AI

Introduction

In previous chapters, I repeatedly used the hypothesis: if software has vulnerabilities, Hacker-AI will find them. Additionally, we should assume that software has backdoors or sleeper code. With Reverse Code Engineering, Hacker AI is flexible enough to find ways to manipulate every system (i.e., any OS, CPU, or device type), steal crypto keys, modify other software, to remove data traces it inadvertently generates. Hacker-AI could generate malware that manipulates the underlying OS to become undetectable and potentially irremovable on every device it visited. Once it repels late-coming similar advanced malware, it could become the only malware.

Under these circumstances, how can we have security? How can we trust any security tools that we develop? Well, this should be no surprise. It is the result of what we did to ourselves. It is not that devices are vulnerable by accident. I am not saying that it was done deliberately. That is not true, but it was responsible for that. We are even accepting this. We are accepting the trade-off that we pay for security. More expensive is likely safer (which is, BTW, not true). More money into cybersecurity has not improved anything. It is more convenient to leave things as they are because that seems to be the safe solution; at least, no new bugs.

The problem is: our cybersecurity was never safe. Security was always part of a trade-off with other goals: convenience, performance, and, of course, costs - but more: why should we risk regular income from updates or subscriptions? Currently, no single cybersecurity solution could deliver alone a solution. Firewall manufacturers can't solve the problem of threats from the Internet. Antivirus solution providers can't protect us from malware, spyware, or ransomware. Even the Operating system providers cannot make their solutions safe alone.

Well, hacking without permission is illegal. Breaking the security of systems without authorization is illegal. But it's still done. That successful hacking exists is not the hackers' fault; it is a deep problem with the underlying technology. Once hackers are using AI, things can get worse quickly. Who cares about legality if governments or out-of-reach criminals use it?

There is no doubt: Hacking or hacker-AI cannot be prevented or limited by governmental regulation or international treaties/diplomacy. Instead, we must

have reliable technical solutions to counter cyberattacks. We must develop these solutions fast.

We have deeply buried ourselves in a quagmire of problems with security and safety. All we do is ignore them and push them down the road. Single solutions can help, but they are not good enough anymore. Instead, we need solution redundancy in cybersecurity urgently. No single point of failure anymore. The saying that a single vulnerability is enough to invalidate all other measures is (so far) true, but accepting this is very dangerous.

On the other hand, having security or safety in software does not mean we won't suffer any harm or damage from software anymore. Software has bugs; this won't change. Some software has even vulnerabilities, but who cares if they are not usable because of redundancy? Still, software will run in environments without reliable redundancy; all users deserve safe software without vulnerabilities. We should also find vulnerabilities must faster.

Unknowns, i.e., unknown security issues, are often huge problems. We don't understand their danger or scope. We often accept unknown knowns by admitting that we have information, but their existence, relevance, or value has not been realized or fully understood. Another word for this is ignorance. Unfortunately, our culture accepts or accuses ignorance of certain problems because we have so many problems. Everyone knows they are ignorant about something, but being ignorant within security is a very bad sign. If this happens within regular product safety: see you in court.

Also, we have unknown unknowns, i.e., unexpected or unforeseeable conditions or circumstances that pose a greater risk because we don't anticipate them based on experience or investigation. Once problems derived from unknown reasons show up, they are no longer unknown unknowns; they are knowns but poorly understood. The problem is our attitude toward security problems. How long does it usually take that security problems to be fixed? It depends on how serious these issues are being taken or what are the possible ramifications of a security vulnerability. These assessments could be wrong. Instead, we should be much more aggressive and hawkish about security problems.

Solution Components

Current cybersecurity is overwhelmed with the challenges it has to face. It has to deal with different threat scenarios, and users don't want to be stopped by security's limitations. Additionally, security gets a lot of lip services, i.e., how important, some say even essential it is; but it loses (often) in trade-offs.

Due to the lack of security, people and businesses suffer hundreds of billions of dollars in losses worldwide. Unfortunately, there is not a lot of critical self-reflection in cybersecurity. Business is booming, and critique is deflected and turned into blaming others.

Our goal should be **proactive, preventative, independent, and redundant cybersecurity**. Most security measures must be low-level and unnoticeable to regular users; otherwise, these methods will be attacked. We should aim for redundancy of independent security, protection, or detection methods that can be called "**security overkill**". Still, it must be unnoticeable to users to be acceptable.

Many architectural mistakes were made around security in the past - they will be mentioned below. Although it is not perfect, proposed solutions must fix the security problems via software updates first, then with hardware retrofits (next) - before we have products with security by default.

Some were asking, why bother about past products? We have a huge problem with (unprotected) legacy devices - we depend on them, and the change to a more secure cyberspace would happen too slowly if we ignore legacy devices and systems.

Unfortunately, a single solution cannot solve the promise of proactive, preventative, independent, and redundant. However, I believe the proposed solutions are easy enough to deploy and acceptable within the existing IT ecosystem.

The most significant change is (A) self-regulation among developers. Developers need to be accountable for their developed apps, done with (B) hashcoding, without exception. Hashcodes can make software easily identifiable, independent of their names. When software codes are registered, they are linked to real people with reputations. The next proposal is Separation (C): regular operations should not impact security operations - no mixing or commingling. (D) Keys are protected from being stolen by not allowing them to appear in cleartext anywhere. (E) Crypto devices or security components are protected from misuse or modification by having them watching each other. And finally, (F) low-level security must be highly automated, i.e., we must react to global Hacker-AI threats instantaneously; humans involved would slow us down. Now, let's go into each proposal more deeply.

(A) Making Developers Accountable

Using software is not so different from taking medicine prescribed by medical doctors or legal advice from lawyers. These occupations and others, e.g., in financial services, have self-regulatory rules protecting the public from rogue pretenders claiming professional reputations.

Often, governments demand regulation, but they are not doing it for the business sectors. Governmental involvement could give official approval too much importance. Instead, peers within a business sector set and apply rules because they have the expertise to know what is normal or an unacceptable anomaly. Most business sectors have an intrinsic interest and common motiva-

tion to keep their occupation free and clear from scandals. If they are too forgiving with bad apples, then this could turn into some outrage about their complicity quickly.

Self-regulating bodies are kicking people or businesses out, preventing them from making a living in their profession. They do this to preserve their standards, particularly their compliance with their ethical or legal standards. Being expelled from making a living with skills and knowledge for which a rule violator invested years of their life is an impressive wager for doing these services.

Self-regulated business sectors create internal policies and procedures for ethical decision-making, establish codes of conduct, and implement systems for monitoring and enforcing compliance. This effort builds customer trust and reduces the likelihood of legal action against companies. Over time, it helps also to identify and mitigate risks. Examples of self-regulation are industry standards and guidelines.

I propose that every code or software is registered automatically via unique hashcodes before being delivered to customers. This transparency creates indirectly enforceable accountability for intentionally inserting malicious code (snippets). As an immediate benefit: using hidden vulnerabilities (backdoors) or exploits by developers would become risky, reputation-damaging behavior. The transparency of registered software creates both: evidence and deterrence of wrongdoing.

In this process of registering, manufacturers and developers share (voluntarily) additional information on software's relevant capabilities and third-party components, which could enable us to detect suspicious activities easier. 3rd party components providers could quickly notify all impacted developers about their fix.

When developers tell us about security-related features, done via simply answerable checklists and checkboxes, they indicate that they are aware of features that could cause harm or could damage users. Developers are expected to take these disclosures seriously because their reputation depends on that, or software tools within their software development environment will do that for them reliably or support them significantly.

Registering is not about tracking developers' performance, calling out developers for making mistakes, or creating unintentional vulnerabilities. This proposal is designed to create a well-earned reputation based on comprehensive, truthful disclosures and responsive, cooperative behavior in case of problems. The additional work is minuscule compared to the value it creates for all involved.

(B) White-/Gray-/Blacklisted Hashcodes

Every software file (including software library or script) is hashcoded and managed within the context of a software publication. Software is considered a perfect clone; no variations within the executables. Some variations could be

allowed, but these are then variations under observation. Manufacturers or software developers could be consulted if it indicates something more nefarious.

When hashcoding is started, i.e., as a new layer of security, hashcodes are confirmed/validated (as known) via server requests (validation or rejection) and then cached in a local repository as known. Or if there are unknown hashcodes within the context of the software, then the hashcodes are locally stored/flagged as suspicious.

Hashcodes are linked to known software packages; validation can be derived from statistics (i.e., an overwhelming vote indicates that the component is part of the original publication). Hashcodes are not necessarily from registration. Non-registered hashcodes (accepted based on statistics) are called graylisted. If the number of votes is too small to determine if hashcodes are acceptable, they are then flagged for a later round of server-side validation.

The new normal should be that hashcodes are voluntarily shared via registration data coming from manufacturers. These hashcodes are called whitelisted, approved-listed, or simply accepted. The validation server provides for accepted hashcodes software-related (additional) data, which are also cached locally. These additional data are used independently and redundantly (by what is later called watchdogs) to determine if the software was covertly manipulated in RAM to anything else than originally designed. These watchdogs instantaneously report any deviation from known software behaviors.

Blacklisted hashcode or block-listed is hashcoded software that is known to be harmful or known to be manipulated by attackers. No executable on that blacklist can be loaded into RAM anymore; only white- or gray-listed executables are allowed. Most importantly, no unknown code, not even a script, is accepted in RAM or by the CPU.

Therefore, only known and trusted software could (theoretically) exploit vulnerabilities. But, if that is done, it would have severe reputational consequences for the developer or manufacturer doing that. And then, it is up to users or device owners if they accept graylisted software or insist on whitelisted software.

In a closing note on this solution: I prefer the terms white-, gray- and blacklisted hashcodes or software because these colors form a certain unity in their terminology. They are quickly associated with each other, which can't be said when we call black-, block-listed, or white-, accepted-listed. But terms are conventions and are called whatever we agree to call them. I will side with the majority.

(C) Separate Security-related from Regular Computations

Before being provided for execution to the CPU, executables and their data are in a single place: RAM. The OS manages access to data or executables loaded or stored in RAM. The OS is the sole instance providing security methods for all data managed in RAM. The CPU manages security-related software

in different "security rings", but it is still not sufficiently protected against regular software sharing the same memory space. CPU's security ring manages privilege levels (0 most and 3 least privileged) that dictate what operations a process can perform.

Some OS providers claim their system is secure/safe based on the software they are using; unfortunately, that cannot be taken at face value. It is extremely difficult to prove that systems are secure (and words are cheap). Complex systems are not perfect; they have vulnerabilities. We must assume that Hacker-AI will find these vulnerabilities and exploit them.

I propose: Due to the complexity of CPU and OS (and their interplay), these systems should be left alone, i.e., we can ignore them as "the deciding voices" when it comes to security decisions. We leave them as they are, and if the OS or other code wants more than (unaltered) security would allow, then a second, independent, preferably separate (duplicated) instance would stop that. The advantage, we have the old, regular security and an additional separate (incorruptible) security that steps in when we need to be suspicious of the regular code. This measure duplicates the checks (i.e., redo the check by the OS within the regular domain) if security-related tasks are not violating security rules.

So, **we create another layer for security-related features,** i.e., independent, separately controlled access to storage and network features (Executable Watchdog, Content Watchdog, and Network Watchdog - explained later). There are (other) good but not vital reasons that this layer is (potentially later) extended to in-/output features, like mic, video-cam, or other hardware resource-related features.

All software within this security layer, which I call the "Security Domain", cannot be changed undetected. This domain is designed to be "untouchable" by regular/normal software, i.e., software from the "Regular Domain", in which we have the OS software and software supported by the main OS, i.e., very software, including user software and eventually malware.

All security-related requests are accepted (i.e., passed-through) or rejected by this additional security layer. The OS can easily handle situations when executables can't be loaded, or files cannot be overwritten or modified. Because OS should have rejected and not accepted an operation (the security layer rejected it), we have a situation in which software was likely misused based on a vulnerability. This event is not relevant for a system protected by the security layer, but it is important for the other systems that want this flaw is being fixed.

The security layer uses for stored files/apps (locally) cached hashcodes and additional data (voluntarily provided by developers) for accepting it.

This security layer also keeps all software up-to-date by facilitating automatic updates. We must insist that all software is current and fully updated for security reasons.

I visualize the security layer as being guarded by several watchdog components that can't be bypassed, like a bridge. All data between RAM and the storage or network component must go through the cable of the databus - there are no other connections that an attacker can use to bypass the databus. The Harddrive is connected via a serial cable connecting the motherboard with the drive; we could replace the cable with another cable that includes the additional hardware providing this independent (separate/duplicated) security, or this component could even be included in a bridge connector on either side of the serial cable.

The Executable Watchdog protects all executables from being manipulated covertly. With this watchdog in place, we can have more (incorruptible) trust in software/instructions loaded in RAM. The Content Watchdog protects user-generated content, e.g., against ransomware. It is a (redundant) component if some executable acts suspiciously, i.e., wants to do more than reported by the developer within their registration. The network is being protected with the Network Watchdog, which serves as a low-level firewall. This watchdog prevents software from misusing the network for suspicious activities like piggybacking within the data exchange or from spyware.

This security layer, with its watchdogs, could be inserted in an independent hypervisor as a software-only solution. This hypervisor is a super-supervisor below all main OS activities. The watchdog activities in this hypervisor are the same as the separate security hardware versions, but they are in the same main RAM, outside the memory that the main OS can access. The software solution is probably ok. However, I would never bet against an AI. It's much better to have independent hardware for security.

The main problem with every software-only solution is that watchdogs, like every other security software, require unmanipulated hashcode data, i.e., it requires reliable encryption in which crypto-keys cannot be stolen and used to manipulate encrypted or digitally signed data. Because software-based en-/decryption on the CPU can never be trusted, a software-only solution is not permanent. An advanced Hacker-AI can likely breach software-only security because it has stolen (cleartext) crypto-keys; this is unacceptable when security matters.

Unfortunately, I am not sure if additional (software-based) hypervisors and security layers are enough for existing (i.e., current generation) smartphones. Additional research and ideas are likely required to make smartphones more resilient against spyware. I am not saying that this is not possible (yet).

(D) No Crypto-Key in Cleartext

Security and secrecy of crypto keys are considered very important, essential, and even indispensable. Information security and cryptography have a history worth mentioning. They are spin-offs from hot and cold wars. With only a few

changes, cryptography was commercialized for the Internet. When used for the military, malware was not part of that threat environment.

New ideas were introduced in commercial cryptography, but three malware-related problems were not sufficiently solved:

(1) Keys can be stolen (undetectable, covertly) on both sides: sender or receiver systems,
(2) Crypto units/engines could be manipulated (and turned into traitors),
(3) Crypto units/engines could be misused without the knowledge of the owner/user.

And the consequences today: tens of trillions $ of eCommerce revenue and national security depend on commercial encryption. If malware does anything of the above covertly, then we are in deep trouble.

In this section, I focus on the first problem, i.e., that keys can be stolen, and I leave it to the next section to deal with the crypto-units or engines.

To protect keys' secrecy, we must demand that no crypto-key appears in cleartext in RAM or within the main CPU (ever). I repeat, no key in cleartext outside of controlled, protected units - Never, ever.

Due to many OS vulnerabilities, keys processed by unprotected CPUs/OS must be considered compromised. All crypto processes must happen in components outside the main CPU and OS. No key should be allowed to be shown in cleartext. That means we must have separate/independent encryption and decryption units (EDU), i.e., crypto units/engines, with protected key storages, i.e., keysafes. Together, crypto-units with keysafes manage all crypto-related processes without exposing the used crypto keys to the outside.

Additionally, every key use must be tracked reliably (e.g., via the number of uses or checksum of exchanged data), enabling us to detect misuses independently - redundantly. A single protection method will not suffice if we deal with Hacker-AI. Redundancy to a level of "security overkill" should be implemented on a lower component level, unnoticeable for users - fully automated and invisible in its details.

PKI (Public Key Infrastructure) provides secure communication and digital identities by using a combination of public and private encryption keys and a central authority for issuing and managing those keys. The public and private keys are different. Currently, public keys in PKI are announced via unprotected certificate files so that they can be inspected by humans visually. But no one cares about that. How often in the last 20 or 30 years have humans manually inspected crypto-key details in a PKI certificate? These operations are automated; the mathematical operations are standardized. There is no need to debug basic algorithms used in en- or decryption or being tested on real keys. Humans have no business seeing any key. Providing public keys in cleartext is a severe mistake.

Therefore, I propose an intentionally incompatible PKI+ in which public keys don't appear in cleartext. Currently displaying, these keys are used in PKI

to identify owners of key pairs. They can say, see, I can publish my public key with my name - use my key, and you can send me messages that only I can read.

I propose additionally that all keys are referred to via their computed hashcodes; this is enough to associate keys with devices and users. Making this connection is based on the history of usages in which misalignments between identities and keys/hashcodes are easy to detect because we deal with reliable hardware (with unique keys), and not easy to copy keys as in data strings. Also, hashcodes don't spread info on any public/private key pair - their key size could be significantly reduced.

Only hardware-based crypto-units (the mentioned EDUs) have keys (from their manufacturing process) capable of creating protected connections with key directories - no software version of the EDU could have this information, except it would use compromised keys.

Additionally, instead of using a single keypair for popular service, as we do right now, we could have 100s or 1,000s of Multiple Equivalent Secret Keys. There is no scarcity in creating more keypairs and no penalty for having more. On the contrary, different crypto units/devices could have different small subsets (of these equivalent secret keys), and the entire subset of keys is used in sending/receiving session keys. Only the original crypto-unit/EDU can know all secret keys. It would lose or change keys when the EDU hardware is probed physically. The hardware design of the EDU would destroy all other keys after a single key is extracted in cleartext forcefully.

Without hackers gaining covert access to session keys or public/private key pairs, we can be much less concerned about the manipulations of exchanged data (e.g., via a man-in-the-middle attack - this is just a sidenote for some experts who know what that means).

If an adversary could steal keys, they are automatically deactivated seconds after detection, replaced with new keys but kept usable as honey-pots. If keys are only suspected to be compromised, they can be flagged and turned into a honey pot. Key replacements are cheap; they are automatically replaced without letting anyone know. Even if key safes are damaged, they could be restored using the cleartext hashcode provided from a backup.

(E) Interguarding Multi-Unit Security

This proposal goes much deeper into how software and hardware must be developed (for security). We built technology up from components; these components are based on reusable patterns that are organized in what is called a component architecture. This approach is designing and organizing individual building blocks (components) in a modular and reusable way.

Each component is a self-contained unit of functionality that communicates with other components through well-defined interfaces. This method clarifies and separates concerns and makes it easier to change or replace individual parts

without affecting the rest of the system. So far, this approach has enabled us to develop, maintain, test, and scale our technology reliably.

The current best practice in component architecture is to have the least number of (hard- or software) components with a generic purpose; ideally, it is one. This philosophy is applied in, e.g., encryption, where one component is sufficient for all crypto applications. There is only one TPM (Trusted Platform Module), or there is usually a single crypto card with hardware storage for private or session keys.

However, if an adversary can stealthy access or interface with these components, it could covertly impersonate the owner/user and misuse this component without having resistance. There is no independent detection by other components that a misuse just happened. This problem is also known as the API-Problem - API stands for Application Programming Interface; it refers to the problem that it can be very difficult for the crypto cards/units to determine whether a request (i.e., use of encryption keys and digital certificates) is legitimate. Crypto cards rely on secure communication and authentication to ensure that only authorized parties can access them. However, these channels, mechanisms, or the request's source can be compromised through malicious attacks. That is bad news for owners of crypto cards; they are on the hook for damages done by misusing these cards.

But if we have multiple components, each having an autonomous and independent OS specialized in specific tasks, they could watch/inter-guard their neighbor instances for covert changes and misuses. A single compromised instance could be prevented from reporting that it was manipulated, but other instances could be enabled to detect/report these anomalies automatically. These validation events (like tripwires) could trigger hidden inspections if the crypto-related code were manipulated, how this attack was done, and which components must be watched or considered compromised or insufficiently disclosed by its original developers.

So, with multiple crypto-units/EDUs independently watching each other if the right/allowed software is used, they create a local network of trust among its units that an attacker must overcome.

Secure updates of security components are based on Multi-Unit Security and a public countdown involving a large group of human experts. We must ensure that (unauthorized) software updates can't happen covertly. This can give us the confidence to change, fix, or extend security device features. With this new Multi-Unit Security approach, software changes are much easier, safer, and more transparent than the handling of changes to immutable software within microcontrollers, which requires a small group of experts to replace (secret) software within new hardware components - while the old controllers with their software remain vulnerable on the old systems.

Lets' go back to the second and third malware-related problems that commercial cryptography must solve, i.e., (2) Crypto units/engines could be manipulated and turned into traitors, and (3) they could be misused without the knowledge of their owner/main user. Multi-Unit-Security can be designed to detect and prevent manipulation of its locally connected units. However, covert misuse (i.e., (3)) of crypto components is more difficult and less reliable to detect. It depends on the context - we may not always prevent it, but we could detect it.

Crypto-key and crypto-unit misuse is a serious issue for humans: we must know the source of a request, or we have the potential for misuse. We will have security breaches when we don't have a redundant overkill of multiple independent security measures. All apps are hashcoded, and we could make someone accountable for inserting malicious code into their software. We may not be able to prevent a bank robbery, but we will know who did it. Then we can include an additional barrier that can prevent reliably that this doesn't happen again.

Crypto-key- and crypto-unit-security require an architectural overhaul based on these new paradigms. The security must be proactive, i.e., no keys in cleartext. Also, it must be preventative, i.e., all relevant units must check each other for attack signs continuously and randomly. These measures must be independent, i.e., within physically separate components, and finally redundant, i.e., detecting if compromised keys or misused crypto-units were used anywhere - which would trigger an automated investigation without human involvement to narrow down what/who was or could be involved and then who did it.

These kinds of security measures seem to be overkill. But that is what security demands, particularly if we are dealing with a Hacker-AI capable of creating attack methods that will likely be beyond our comprehension. We cannot have a second security overhaul to accommodate threats from an artificial superintelligence that is doing system hacking much better than the most capable intelligence service.

(F) Automated Security

Humans are usually the weakest link in security. Human involvement is sometimes dangerous because humans could be deceived, blackmailed, or turned into traitors. Frankly, humans should have no business being involved in low-level security.

Secrets must be protected from humans, with no exceptions for special roles. When security, protection, and detection processes are automated, it is easier and more reliable to protect security measures from covert modifications. Any attack on automated security can be detected as a severe anomaly that must be investigated (automatically) to determine patterns or possible vulnerabilities, or undisclosed capabilities.

Cybersecurity automation provides speed, efficiency, scalability, consistency, cost-effectiveness, comprehensive coverage, and predictive capabilities to detect and respond to security threats.

Automation allows faster and more accurate detection, including response to security threats; it reduces response time to mitigate a security breach and minimizes the impact on devices and their data. We can handle larger events and keep up with the ever-increasing volume and complexity of threats. We can enforce security policies and procedures more consistently and reduce the risk of human error. Automated systems do expensive and time-consuming tasks 24/7; they can also do repetitive tasks like monitoring processes or analyzing large amounts of data more efficiently than humans. Automation will allow us to use machine learning to identify patterns and anomalies in data and to predict and prevent potential security breaches before they occur.

Full low-level automation is required, mainly because Hacker-AI attacks could happen quickly. The defender's reaction speed will determine whether camouflaged malware from Hacker-AI continues to be accepted or blacklisted before it can form a beachhead.

But this automation will not replace humans in cybersecurity entirely. Human expertise, judgment in the field, decision-making, and human oversight are necessary to manage and interpret results. Cybersecurity teams will need to collaborate with automated systems in complementary ways in helping to enhance our security posture and to make the most effective use of both human and machine capabilities.

But some information and processes must be outside human reach for the security and protection of the humans involved. There might be a few tasks within incidence investigations where humans could provide value. However, having oversight, having the tools to enforce a final verdict in big decisions, and allowing or denying validated updates to security components are competencies humans can't give up.