2028 - Hacker-AI and Cyberwar 2.0+
(Securing our Future: Proactive Resilience through Separated Security Measures)
ISBN: 9798375252568
Copyright © 2023 Erland Wittkotter, Ph.D. All rights reserved.

12. Countermeasures: Understanding Why they Work

Proposed Solutions Applied to Problems

Within the second chapter, I mentioned some deep-rooted problems with our software-based technology that resulted in a list of challenges we must face, or we have no cybersecurity. Let me repeat this list briefly as a reminder before I will discuss them in the follow-up in the context of our proposed solution.

(1) “**Software is invisible**”: Software can only be seen and validated indirectly; it requires trust in the entire ecosystem to trust any part. Rephrased: “Software must be trusted (blindly)”, and someone else “will (hopefully) see the problem”.

(2) “**Software is covertly modifiable**” Emphasis is on “covert”; detection of modification is not reliable. We have too many blindspots with mutable software where covert/temporary modifications can happen.

(3) “**Every tool/component could be compromised**”. We must question our trust in whatever we use. Rephrased: “Any software can be dangerous.

(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, 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 considered a critical core problem within cryptography yet.

The above list of problems can be divided into four categories: software, attacker, defender, and crypto. The first three problems are software-related, and the next four are attacker-related. Then we have four defender-related problems (8 - 11). The last problem is infrastructure, but mainly crypto-related. These categories or the above list do not claim to be comprehensive but a start.

If we dare to tackle the above cybersecurity problems, we must do this comprehensively with as many tools and with as many problems solved. Although redundancy could help us to avoid damage from failures in one measure, the weakest component within security could potentially waste all our progress.

Next, I want to show that the proposed security solutions make security as simple as possible but not simpler. Additionally, I will show how the danger from Hacker-AI-generated malware, Cyber Ghosts, and Cyber Devils could be solved technically. To prevent malware from Hacker-AI, we must solve the above issues and turn technology in favor of cyberdefenders.

Unfortunately, cybersecurity or cryptography does not contribute enough solutions yet, and quite bluntly, many or most (?) security professionals seem to have given up on providing help for these issues. Repeatedly, hacks are sneaking through the cracks, and frankly, I cannot blame anyone dealing with security for taking the proposed solutions with a grain of salt. However, I will show that they make many / all of the above issues irrelevant or manageable.

The discussion below may not be enough, but it is a start. I expect to hear or read compelling examples if the arguments are wrong. Being doubtful if it works is ok, but this should not be an excuse to do anything. Hacker-AI and Cyberwar 2.0 are real threats that we must deal with sooner than later.

In general, all proposed security tools are about uncovering covert security-relevant activities and identifying the source of the attack. Stopping and identifying them is accomplished using cached hashcodes for the app and key tracking. Known hashcodes are accepted when they are not blacklisted. Unknown hashcodes are rejected by default and potentially investigated.

Our basic assumption behind the proposed security is that in-transparency and covertness around (critical) operations have led us to our vulnerabilities. Therefore, we should push for more transparency. Once attack patterns that harm or damage users become detectable, they can likely be handled with business or legal rules. Security rules can be adapted over time to include exceptions, but users should confirm deviations from default security-related operations/rules (incl. settings) or be informed.

In the following, I discuss how the proposed solution components affect the discussed problems. I restate the problems in light of the new approach to cybersecurity.

Software-related Issues / Solutions

I don't want to make the solution more complicated than they are. So I get straight to the how and why it will work.

(1) "Software is invisible, but becomes identifiable".

We can only allow white or graylisted software in RAM; this requires hashcoding and the registration or statistical analysis of all software (including scripts) to determine which software we know as legitimate. Once we know that software comes from a (known) source, we could make the developer or manufacturer accountable for their malware or for using an exploit.

With hashcodes, we can reliably gather additional information about executed software; this also applies to graylisted apps. Over time, software becomes more transparent and, thereby, more trustworthy, i.e., it is only doing what we expect it to do; the software does not surprise the watchdogs with hidden (security-related) features that would raise flags.

With a hardware-based Executable Watchdog, we can be sure that even the hypothetical Cyber Ghost or Cyber Devil can't avoid hashcoding. Every code must inevitably go through a bottleneck, the databus. The databus contains a separate component that uses trustworthy code from the security domain to analyze every executable. A Cyber Ghost cannot modify the security domain with its regular code, and all hashcodes are reliable because we can trust encryption and digital signatures with the use of keys that can't appear in cleartext outside protected crypto-units.

We only need to create additional measures to eradicate malware from possible hideouts like BIOS, UEFI, or other NAND gates used as memory components (like flash memory cells) within microcontrollers on the motherboard or within the storage, network, audio, or videos cards. We know these possible locations, and we will likely have, sooner than later, sufficiently good tools for cleaning these locations.

With hashcoding, we could make software visible and identifiable whenever it is required to know that it has not unexpectedly changed.

(2) “Software is (not) covertly modifiable”.

If the hashcodes derived from the executables are checked before every use, i.e., before apps are loaded into RAM, covert modifications to these apps are detected immediately with dedicated/separate hardware (i.e., with the Executable Watchdog). Multi-Unit-Security facilitates overt, legitimate, and non-obstructed modifications and updates, but it would create an instant alarm if the watchdog is manipulated so that the watchdog would let in some manipulated software into RAM.

In software-based security solutions, decisions on accepting hashcodes in RAM are done via the main CPU/OS; there is the possibility that Hacker-AI/malware could have found a way into RAM and let other modified software in, or the malware could modify/compromise other apps in RAM.

If we have the hashcode generation and security decisions related to hashcode status (white/gray/black or unknown) done in physically separated components operated with OS-independent software (i.e., separate security hardware within the non-bypassable databus), then this problem of undue influence by code running on the CPU is solved.

It is possible to include the hardware as a retrofittable bridge component within the databus, e.g., by changing the cable or via a bridge-connector on either side of the original cable, and the PC or laptop would be safe. This retrofit is not doable with existing smartphones.

(3) “Every tool/component could (still) be compromised, but we can stop it and know who did it”.

Hacker-AI could still try to manipulate development tools and get backdoors or sleeper code in published software. Developer tools or third party-components can be compromised and infect other software. These problems will be fixed as soon as they are discovered. Over time the number of occurrences of these types of attacks will shrink; old backdoors are discovered, and new ones are increasingly more difficult to create.

Also, including malicious features or backdoors into products are intentional decisions by software developers or manufacturers. Even later, if malware features are discovered, the reputational damage could be catastrophic. Being proactively transparent about their mistakes could be their only way out.

Every late modification to code is detectable via hashcodes. It should not be easy to quietly cover up features with intentional and covert damage potential. With hashcodes associated with archived binaries, old software and their chain of modifications could be analyzed anytime (later).

Also, software vulnerabilities are of no concern as they require detectable exploits. If vulnerabilities are used, this is a serious violation of trust with reputational consequences. The registration process may allow a confidential coming-clean confession. In time, the software ecosystem will become significantly less compromised and less vulnerable due to the steep consequence of developing or using exploits.

Attacker-related Issues / Solutions

I could go directly to the main reputational point of why nobody dares to be an attacker: it would likely end someone’s career. But still, because of redundancy and because I raised other attacker-related issues, I want to show how they are being solved.

(4) “Attacker chooses methods and timing - but has no benefit from that”.

Having a first-mover advantage is not enough anymore. Defenders can create within their separate security domain honey-pots or tripwires that attackers cannot know or systematically explore without the risk of being detected in probing these security / detection methods.

If they explore weaknesses in an attacked system, this is done by white or gray-listed software. If this exploration of features or vulnerabilities is detected as suspicious, the software will be blacklisted immediately. However, these measures are bypassed within developer’s environment, where temporary registration data are accepted as real whitelisted data, and no suspicion is raised.

Sooner than later, compromised gray or whitelisted software will raise a flag when it does anything suspicious, like elevating the permission rights of software components, opening unknown file types, changing the attributes of files, etc. Everything that the software is not normally doing or what was not disclosed within registration is an anomaly. Anomalies trigger automated investigations of suspicious software. The first time attack software uses its attack capabilities, it becomes exposed to automated scrutiny - that is not a first-mover advantage.

(5) “Attackers know more (about vulnerabilities) - but they won’t dare”.

Because hashcoding detects modified software and exposes the exploitation of vulnerabilities, knowing more is insufficient to gain an advantage in an environment that expects white- or gray-listed software. The attacker’s only advantage is having secret knowledge of the availability and features of backdoors or sleeper code within unknown software. But using these features might make them immediately known.

Also, to succeed, attackers must first risk their anonymity and reputation; Rebuilding a reputation is time-consuming.

(6) “Attackers can adapt to (known) methods of detection - but can’t bypass it”.

Attacker changing their appearance creates a new unknown hashcode which is rejected by default. Which data the separated security layer uses to detect malware or anomalous white- or gray-listed software uses is unknown to attackers.

Bypassing detection is futile; the same applies to removing undetected data traces. Attackers can only use regular processes without access to the security domain in which data traces are generated. What is collected cannot be accessed or modified by software within the regular domain executed on the CPU.

Cyber ghosts could theoretically bypass software-only protection via updated software. But it is doubtful they could bypass separate security hardware components watching each other for suspicious misuse or modifications.

(7) “Attacker can (not) get away unidentified”.

Attackers are identified when they register their software. Remaining anonymous is making them potentially suspicious. If users accept new grey-listed software, it is up to them to accept that risk. Attacker’s presence and involvement will be discovered because there will be an unacceptable or suspicious event leading to damage or harm to the device owner or user.

Defender-related Issues / Solutions

I want to show that the defender position has significantly changed once the proposed solutions are used in cybersecurity. Defenders will have an edge, but this edge should not be gambled away with overconfidence. Defenders are vulnerable as regular people; they could be blackmailed or bribed and turned into traitors. If a small group of humans remains important deciders within the defense, they could also become unwilling targets.

(8) “Some Secrets are unreliable in defense - others can be made reliable”.

Secret data, including encryption keys and operational status settings (i.e., is a security measure active or dormant), are made so secret that no human can know them, even when they are trying hard. The secret of which data were generated or used within defense for detecting anomalies is only shared after it was used, i.e., when it becomes essential evidence.

Operational secrets that no human or process within the regular domain knows is the only reliable secret.

Intentional uncertainty can deter attackers from probing security covertly or systematically because it could be suspicious, indicating preparation for an attack.

(9) “Defenders can (not) be compromised”.

Security is proposed to be automated and non-modifiable via decisions.

Additionally, it will be humans who decide how much risk they accept. If there is a risk of blackmail or bribing, this person has too much responsibility. It is much more difficult to attack 4, 5, or more if anyone could sabotage a decision covertly.

<u>(10) “Software output could still be faked (late), but we generate irrefutable evidence”.</u>

Some transactions are known to be harmful, and others are critical for attacker preparation. With independent log data within the security domain around regular transactions, the details of these logs are unknown to attackers and human operators. However, commercial transactions are made detectable to the security domain so that these data can be stored or logged as irrefutable evidence in case of any discrepancies in these transactions.

The software in the security domain would log enough data to discover fake output from the past (via replay) or the present via detecting modified software within the software stack. The Executive Watchdog will trigger an in-depth investigation by providing an automated (full) report to a system dedicated to these reports and follow-up investigations.

<u>(11) “Complexity is an enemy of security - we use / deploy simplified, dedicated systems for security”.</u>

Our current technical ecosystem is extremely complex. Many different old and new hard and software components are continuously updated, giving these systems more functionality. With thousands of technical standards, systems are integrated. Our IT systems use multiple technologies in parallel or are built on top of each other. The technology uses hundreds of programming languages. We are probably in the thousands when we are also considering major versions.

However, by separating a standardized, independent security domain from regular code executed within the main OS and CPU, i.e., the regular domain, we have a simplified security environment in which only security-related tasks are executed. Attacker’s regular code has no access. Additionally, the separate security domain can be updated with tightly controlled updates. At the same time, Multi-Unit-Security guarantees that only trusted, standardized, i.e., allowed code is being executed in the security domain. Every deviation from the standard is easily detectable as an anomaly. This simplification is much easier to defend than a complex commingled code environment with security and regular code.

Crypto-related Issues / Solutions

I return to the three problems of commercial cryptography: 1. Keys can be stolen covertly / undetectable, 2. Crypto units / engines can be manipulated, 3. Crypto units / engines can be misused without detection.

<u>(12) “Crypto-Keys / Units are protected - crypto-misuse is detectable”.</u>

Because keys are never allowed to be shown in cleartext, systems exporting keys are flagged. All devices of that type would then be excluded from receiving protected/secret keys. Additionally, compromised keys are replaced automatically without providing any hints that this has been done. All potentially compromised keys are used as honey-pots. Using protected keys by non-protected encryption/decryption hardware can be detected reliably via (aggregated) data that is being stored about the key usage.

The misuse of crypto components is detectable because multiple security or crypto-key units watch each other if being misused. Covert misuse cannot happen. Still, misuse attempts are being traced to attackers and their tool use. Malware has no access to keys as they are all managed and processed within the Security Domain. Also, the use or misuse of the Crypto units is being stored/logged via algorithms inaccessible to software within the regular domain.

Hacker-AI-related Issues/Solutions

Hacker-AI activities can proactively be stopped because of the strict separation of regular computation from security-related features. Once we can stop unknown apps, i.e., apps with unknown hashcodes, from being loaded into RAM, we are taking away malware's foundation from existing on a device. Still, additional proposed measures as redundancies malware slipped through the cracks.

Furthermore, we will have different situations for each device because we initially have only a software-based and not a (retrofitted) hardware-based security solution. This is problematic for smartphones and other devices for which we cannot provide retrofits.

Let's repeat what we need to accomplish. The proposed security solution must protect us against malware generated by Hacker-AI, which has the following main capabilities: (a) finds vulnerabilities or actively create vulnerabilities for rights-/permission-elevation, (b) steals or misuses encryption keys, (c) hides as a Cyber Ghost, i.e., circumvents detection methods, and (d) making itself irremovable on a device.

Methods against the first three capabilities were already discussed above and should not be repeated here.

The irremovability (d) is an issue that has not been addressed or explicitly discussed here. If Hacker-AI tries to make itself irremovable after the security components are deployed, then Hacker-AI's malware is likely rejected by one of the redundant security measures due to unknown hashcode or failed access to the security domain, etc. Additionally, it would be rejected when trying to create a beachhead or find a Cyber Cradle. Gaining sufficient information about systems, i.e., probing the relevant security, is likely very expensive for a Hacker-AI as it would need to waste or burn through many vulnerabilities, attack tools, and methods, which are revealed to cyberdefenders after their first use.

However, what if a Hacker-AI is already on a device or has left a private backdoor on the system? The problem is that we can't give a definite answer if we have only software-based security. We must assume that this malware with Cyber Devil-type features will relentlessly try to sabotage all new tool components and prevent the separation of security and regular operations. Without additional hardware, we won't be safe. Even with new hardware, we should be very cautious if Hacker-AI has not left some low-level surprise backdoor within our security layer.

In developing the proposed security components, we must assume that Hacker AI already exists and is active, although it may not. This scenario is discussed as being too late in developing or deploying security measures which are done in the next chapter.

In short, if Hacker-AI malware is already irremovable on devices, then we have very little chance of getting it removed with software-only security measures. Only hardware (Executable Watchdog) as a separate component within the databus could give us control back if we would also clean all possible hideouts and ensure that we will have deep hardware-based backdoors in our new security domain.

Still, the most important solution component will be our hardware-based crypto-key secrecy and multi-unit crypto-/security protection against misuse. If these components are available, we may have a chance to get full control back over the CPU, RAM, and our hardware-based security domain, even if it was compromised initially.

Cyberwar 2.0 - related Issues / Solutions

Hacker-AI-based malware is an essential component in Cyberwar 2.0. As soon as malware cannot be used on attacked devices, perfectly executed Cyberwar 2.0 or Cybercrime 2.0 scenarios are no longer a lingering threat. The risks for assailants can be increased, and non-technical defense measures could make sense.

Cyber Reconnaissance via smartphones becomes much more difficult. If companies (operating a widespread smartphone app) would collaborate with an assailant, then getting reconnaissance data via updates in their software is still possible. The same must be said about allowing attackers to have Cyber Beachheads and access to rights or permissions elevation on these devices. But the problem is that the assailant and the collaborating software manufacturer could be caught much sooner. Their actions would likely leave data traces they cannot remove because of the separation of security and regular domain.

We won't be able to prevent an assailant from preparing for Cyberwar 2.0 via building a tech library of more vulnerable legacy devices and tech simulators testing and training their attack tools. We also need to expect that an attacker would work on a Cyber Patsy Designer and Attack Synchronization Management to be at least prepared to simulate different Cyberwar 2.0 scenarios and deflect the authorship for cyber operations to others.

Still, a determined assailant could learn from the concept of Cyberwar 2.0 and use direct access to countries' citizens to get collaborators for doing tasks manually that could lead to significant problems in the continuation of a government or the preparation of military defense actions. If a Cyberwar 2.0 scenario without full Hacker-AI capabilities is still a threat, i.e., could still trigger a government overthrow or regime change, it cannot be answered without knowing additional non-technical (political/legal) defense measures.

Developing and deploying countermeasures against Cyberwar 2.0 requires an understanding of country's vulnerabilities from battlelines that could go through the entire civil society. Without a broad deployment of technical countermeasures against malware from Hacker-AI, the threat from operational Cyberwar 2.0 measures will remain.

We could see an arms race between defenders and attackers. Still, with proactive, preventative, separate, and redundant cybersecurity solutions, we could give cyberdefenders a fighting chase. However, vulnerabilities from legacy systems will remain a significant problem for many years.

New Cybersecurity Paradigms

Seeing problems or issues slightly different has a significant impact on conclusions or motivation for actions. Problems can often be ignored or reevaluated because we view something differently, i.e., based on new paradigms. Unfortunately, some current cybersecurity-related ideas and paradigms are detrimental to improving security.

The following list should serve as a start:

(1) Do not Trust CPU/OS.

Cybersecurity knows that CPU's and OS's complexity are the core reasons for most vulnerabilities. But shared opinions are seemingly insufficient to accept that this has consequences for IT's design and architecture. Physical separation of CPU tasks in security-related and regular tasks will make a huge difference. If we don't trust the CPU/OS, then we should not use it for security.

Security/control operations are (usually) rigid/static, while all other (regular) operations are versatile and dynamic. Separating security is similar to circuit breakers or fuses in power distribution, i.e., it has little complexity in its on-/off feature.

Security operations can be protected much easier when security is not commingling within the main RAM. As a result, regular algorithms would have no access to security. A databus cannot be bypassed, which makes it a near-perfect location for a circuit breaker, i.e., separating and providing independent security components.

(2) Regular Local Code Validation.

Once installed, software is often considered safe/secure. But software can be modified or reconfigured covertly without being detected as a malicious act.

Instead, we could hashcode all local executables and enrich them with data inferred from statistics, which makes them graylisted, or via voluntarily shared registration data from manufacturers, making these hashcodes whitelisted. Apps with unknown hashcodes are rejected by default.

Once additional data are cached locally; they can be (regularly) used to detect deviations from known software details for instant reporting.

(3) Software Developers must be made Trustworthy.

Medical doctors, lawyers, and financial advisers already have self-regulating rules providing a minimum level of quality control for the public. The same should apply to software developers and manufacturers. We should know at least that they are not cybercriminals. Additionally, we should be able to make them accountable if they cross any red line.

Developers' truthfulness with (independently validatable) safety-relevant product disclosures is associated with reputation. Information shared on their software could help significantly determine what threats or surprises we should not expect as regular/accepted software behavior. The developers and manufacturers are responsible for feature integrity and reasonable security-measure within the disclosed security-relevant operations.

Problems within disclosed product features are accepted as (normal) bugs. We don't assume maliciousness until we see evidence (like exploits or backdoors).

Deviation from disclosures would automatically raise suspicion. Deceitful exploits in apps would ruin a developer's reputation, while creating vulnerabilities accidentally is normal, harmless, and would be ignored as a problem to a developer's reputation.

(4) Preventing Key-Cleartext Disclosures.

Adversaries determined to steal keys could modify via reverse code engineering key processing CPU/OS software. Therefore, key secrecy should mean the main CPU does not process crypto-keys; only protected CPUs are permitted to process secret keys.

Every key that appears or could theoretically appear in cleartext on the CPU must be considered compromised, and all hardware devices that could (theoretically) reveal protected keys must be flagged as unreliable for processing secrets.

(5) Establishing Multi-Unit-Security.

Device components are not independent of the OS. Therefore having security components interguarding each other would currently be useless. According to generally accepted design principles, fewer components are considered better than more, but isolated units are prone to misuse.

However, if we have independent Multi-Unit-Security, we could use it to have reliable security units watching each other if any among them is or was modified.

(6) Security Execution / Detection must be Automated.

We should distrust provided security if humans are directly involved in any non-high-level or operational aspect of security. Only independent automation guarantees reliable rule execution. Humans should be prevented from making (security-related) exceptions.

We can use proactive and preventative conditions for rule violations or damage detection if we know what to expect. All automation methods / rules are protected against (covert) modifications, reconfigurations, or updates.