Downgrade Attacks Using Windows Updates | SafeBreach

By Author: Alon Leviev

Archived: 2026-04-06 01:19:17 UTC

Downgrade attacks—also known as version-rollback attacks—are a type of attack designed to revert an immune, fully up-to-date software back to an older version. They allow malicious actors to expose and exploit previously fixed/patched vulnerabilities to compromise systems and gain unauthorized access.

In 2023, for example, the BlackLotus UEFI Bootkit employed a downgrade attack. The malware downgraded the Windows boot manager to a version vulnerable to CVE-2022-21894 to bypass Secure Boot. BlackLotus was then able to disable other OS security mechanisms and gain persistence in impacted systems. The BlackLotus UEFI Bootkit sent shock waves through the cybersecurity community, as it was capable of running on fully patched and up-to-date Windows 11 systems with Secure Boot enabled. While Microsoft patched Secure Boot extensively against downgrade attacks, I couldn't help but wonder whether downgrade protection was added anywhere else in the Windows OS.

My most recent research project—which I first presented at Black Hat USA 2024 and DEF CON 32 (2024)—set out to explore the state of downgrade attacks on Windows. I found several vulnerabilities that I used to develop Windows Downdate—a tool to take over the Windows Update process to craft fully undetectable, invisible, persistent, and irreversible downgrades on critical OS components—that allowed me to elevate privileges and bypass security features. As a result, I was able to make a fully patched Windows machine susceptible to thousands of past vulnerabilities, turning fixed vulnerabilities into zero-days and making the term "fully patched" meaningless on any Windows machine in the world.

Below, I'll first provide a high-level overview of the key findings and takeaways from this research. Next, I'll provide some background information about the Windows Update architecture. Then, I will dive into the research process that led me to develop the Windows Downdate downgrade flow. I will also explain how I was able to downgrade key OS components, bypass Windows Virtualization-Based Security (VBS) UEFI locks, and expose past elevation-of-privilege vulnerabilities in the virtualization stack. Finally, I will highlight the vendor response and explain how we are sharing this information with the broader security community to help organizations protect themselves.

Overview

Key Findings

With a research goal of developing an undetectable downgrade flow for Microsoft Windows, the Windows Update process seemed like the least-suspicious entity through which I might execute such an attack. As I explored the intricacies of the Windows Update process, I discovered a significant flaw that allowed me to take full control of the process. As a result, I was able to create Windows Downdate, a tool that implemented downgrading updates and bypassed all verification steps, including integrity verification and Trusted Installer enforcement.

Armed with these capabilities, I then managed to downgrade critical OS components, including dynamic link libraries (DLLs), drivers, and even the NT kernel. After these downgrades, the OS reported that it was fully updated and was unable to install future updates, while recovery and scanning tools were unable to detect issues.

I then aimed higher and found that the entire virtualization stack was at risk as well. I successfully downgraded Credential Guard's Isolated User Mode Process, Secure Kernel, and Hyper-V's hypervisor to expose past privilege escalation vulnerabilities.

Finally, I discovered multiple ways to disable Windows virtualization-based security (VBS), including its features such as Credential Guard and Hypervisor-Protected Code integrity (HVCI), even when enforced with UEFI locks. To my knowledge, this is the first time VBS's UEFI locks have been bypassed without physical access.

As a result, I was able to make a fully patched Windows machine susceptible to thousands of past vulnerabilities, turning fixed vulnerabilities into zero-days and making the term "fully patched" meaningless on any Windows machine in the world.

Takeaways

The implications of this research are significant not only to Microsoft Windows—which is the world's most widely used desktop OS—but also to other OS vendors that may potentially be susceptible to downgrade attacks. We believe the findings suggest several important takeaways:

- There is a need for increased awareness of and research into OS-based downgrade attacks. During this process, I found no mitigations preventing the downgrade of critical OS components in Microsoft Windows. We believe other OSs may be equally susceptible to similar attack vectors and that all OS vendors must be vigilant against the dangers they pose.

- Design features within an OS should always be reviewed and regarded as a relevant attack surface, regardless of how old the feature may be. The downgrade attack I was able to achieve on the virtualization stack within Windows was possible due to a design flaw that permitted less privileged virtual trust levels/rings to update components residing in more privileged virtual trust levels/rings. This was very surprising, given Microsoft's VBS features were announced in 2015, meaning the downgrade attack surface I discovered has existed for almost a decade. While VBS has become a more popular topic among security researchers in recent years and several great research papers have been published, more research specifically focused on the design of the virtualization stack is needed.

- We believe in-the-wild attacks should be thoroughly examined and expanded upon by researchers whenever possible. The BlackLotus UEFI Bootkit brought the concept of downgrade attacks to the cybersecurity community's attention. Thankfully with this research, we were able to expand on this type of attack before malicious actors did. However, this is not always guaranteed, emphasizing the importance of studying in-the-wild attacks and using them to consider other components or areas that could also be affected.

The Research Process

To kick off my research, I needed to define what the success criteria would be for a "perfect" downgrade attack:

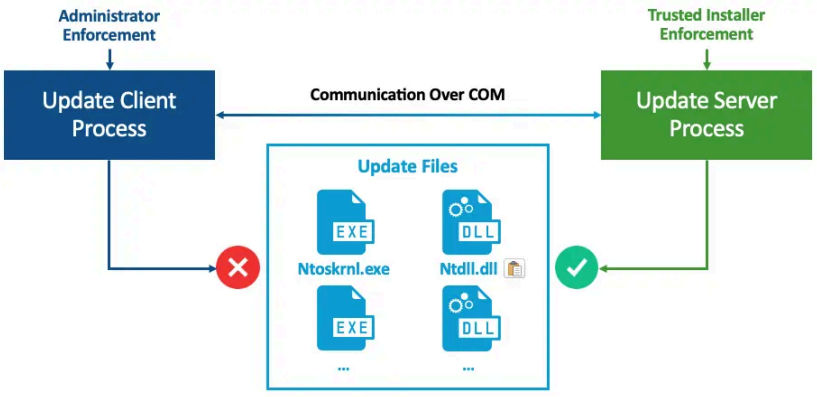
- First, the downgrade must be fully undetectable, so that endpoint detection and response (EDR) solutions cannot block the downgrade. Thus, I aimed to perform the downgrade in the most legitimate way possible.
- Second, the downgrade must be invisible. The downgraded components should appear up-to-date, even if they have technically been downgraded.
- Third, the downgrade must be persistent, so that future software updates do not overwrite it.
- Finally, the downgrade must be irreversible, so that scanning and repairing tools will not be able to detect or repair the downgrade.

With the downgrade requirements well-defined, I then began considering a suitable component to target. What would be the least expected component to perform downgrades? I set my sights on the Windows Update process.

Windows Update

Windows Update Architecture

The Windows Update architecture includes an update client and an update server that communicate over COM, an inter-process communication method on Windows. Administrator is usually enforced on the client side, and Trusted Installer is always enforced on the server side, meaning system files owned by Windows Update are only accessible to the Trusted Installer. As a result, even Administrators and NT SYSTEM are unable to directly modify system files.



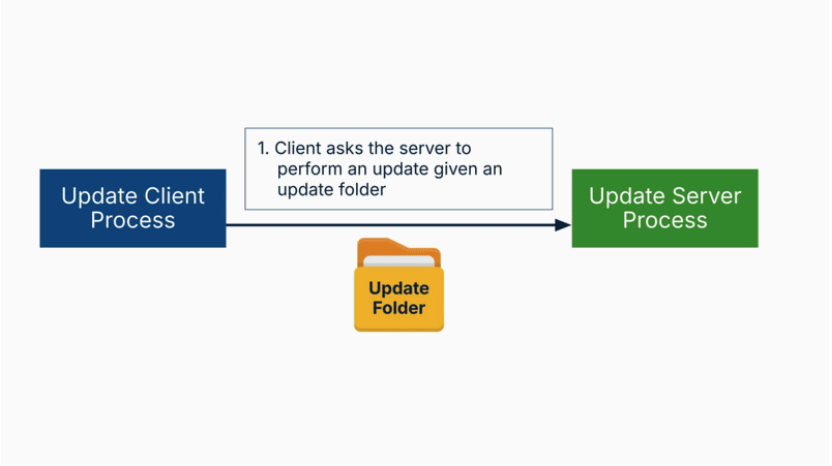
This is where I found the first design problem in Windows Update. Administrator to Trusted Installer is not a security boundary, and there are multiple, working public proof-of-concepts of such an elevation. The Windows Update team attempted to secure the update process by enforcing Trusted Installer. However, since updates are only accessible to Administrators, Trusted Installer is rendered completely ineffective in enforcing access to system files, as one can elevate to Trusted Installer and perform the changes.

Unfortunately for me, Administrator-to-Trusted-Installer elevations are considered malicious and blocked by EDRs, meaning it contradicts my first downgrade principle of being fully undetected. I considered trying to bypass the elevation detection; however, I would have to implement the update process myself, which could still be seen as malicious. The best option would be to find a flaw in the update process that would solve all those problems.

Windows Update Flow

The Windows Update flow includes the following steps:

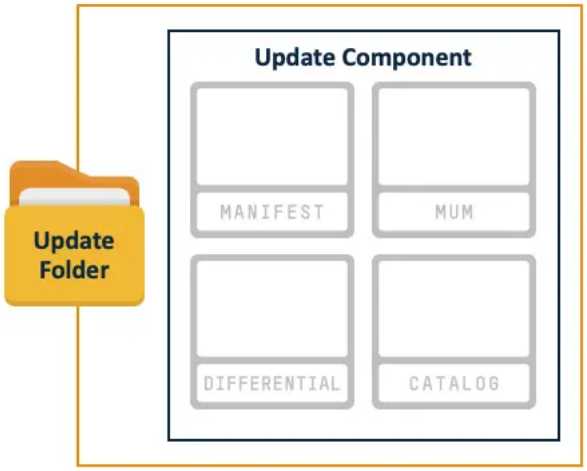
1. First, the client asks the server to perform the update contained in an update folder it provides.
2. The server then validates the integrity of the update folder.
3. Following the verification, the server operates on the update folder to finalize the update files. These are saved to a server-controlled folder, one that is not accessible to the client.
4. The server saves an action list to the server-controlled folder, not accessible to the client. The action list is named *Pending.xml* and it contains the update actions to perform. For example, it specifies which files to update, the source and destination files, etc.
5. Finally, when the OS is rebooted, the action list is operated on, and the update actions are performed during the reboot.



The client only controls the initial update folder. So, I decided to look at this folder first, and see if I could modify it, resulting in custom downgrading update files. As we already know, integrity checks are performed on the update folder. Let's see how well they are implemented.

Investigating the Update Folder

The update folder contains update components, and each update component contains MUM, manifest, differential, and catalog files, as shown below.



- The MUM files are Microsoft Update metadata and contain metadata information, component dependencies, installation order, etc.
- The manifest files contain installation-specific information like file paths, registry keys, which installers to execute as part of the installation, and more.
- The differential files are delta files from base files. A base file plus a delta file would result in the full update file.
- The catalog files are the digital signatures of the MUM and manifest files. Catalogs allow multiple files to be signed at once, instead of having the file we want to sign embed its digital signature. In addition, the catalog files are digitally signed themselves, which makes it impossible to modify them and the files that they sign.

Again, to summarize:

- Only the catalog files are digitally signed.
- The Manifest and MUMs are not explicitly signed, but are signed by the catalogs.

- Differential files are not signed. They also control the final update file content.

Targeting Differential Files

I found the last fact interesting, and wondered if there was any chance differential files were left behind in terms of validation. But, I had no luck there, because the expected update file hash is hardcoded in the manifest. And the manifest cannot be changed, since that would break its signature in the catalog.

Targeting the Action List

Next, I decided to explore the action list. I knew that I wouldn't be able to change its contents, because it is Trusted Installer enforced. But I also knew that the update process is performed over multiple reboots, so I could assume the state of the list was saved somewhere.

I searched the action list path in the registry and found an interesting key named *PoqexecCmdline*. It holds the executable that parses the list and the list path.

I then looked at the security attributes of this key and noticed that it is not Trusted Installer enforced! This would allow me to control all the update actions.

The action list—*pending.xml*— is an XML file that provides the functionality of creating files, deleting files, moving files, hard-linking files, creating registry keys and values, deleting keys and values, and much more!

<POQ postAction="reboot">

<CreateFile path="C:\Windows\System32\Create.exe" fileAttributes="0x00000000″/>

<MoveFile source="C:\UpdateDir\Source.exe“ destination="C:\Windows\System32\Destination.exe"/>

<HardlinkFile source="C:\UpdateDir\Source.exe“ destination="C:\Windows\System32\Destination.exe"/>

<SetFileInformation path="C:\UpdateDir\Source.exe“ securityDescriptor="binary base64:[BASE64-BLOB]" flags="0x00000040″/>

<DeleteFile path="C:\Windows\System32\Delete.exe"/>

<CreateDirectory path="C:\Windows\System32\Directory" fileAttribute="0x00000080“ securityDescriptor="binary base64:[BASE64-BLOB]"/>

<CreateKey path="\Registry\Machine\Key"/>

<SetKeyValue path="\Registry\Machine\Key" name="Name" type="0x00000001“ encoding="base64″ value="[BASE64-BLOB]"/>

<SetKeySecurity path="\Registry\Machine\Key“ securityDescriptor="binary base64:[BASE64-BLOB]" flags="0x00000001″/>

<DeleteKeyValue path="\Registry\Machine\Key" name="Value"/>

<DeleteKey flags="0x00000001″ path="\Regsitry\Machine\Key"/>

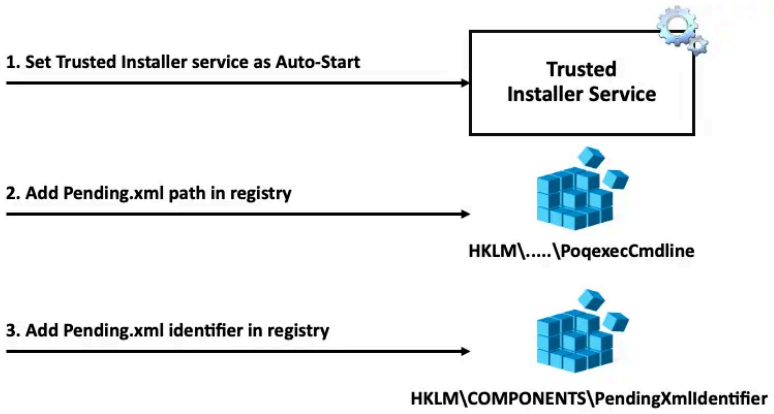
</POQ>

In order to downgrade, I could use the hard-link file action and the source would replace the destination.

<HardlinkFile source="C:\UpdateDir\Source.exe“ destination="C:\Windows\System32\Destination.exe"/>

Introducing an Update

To initiate an update, all I needed to do was:



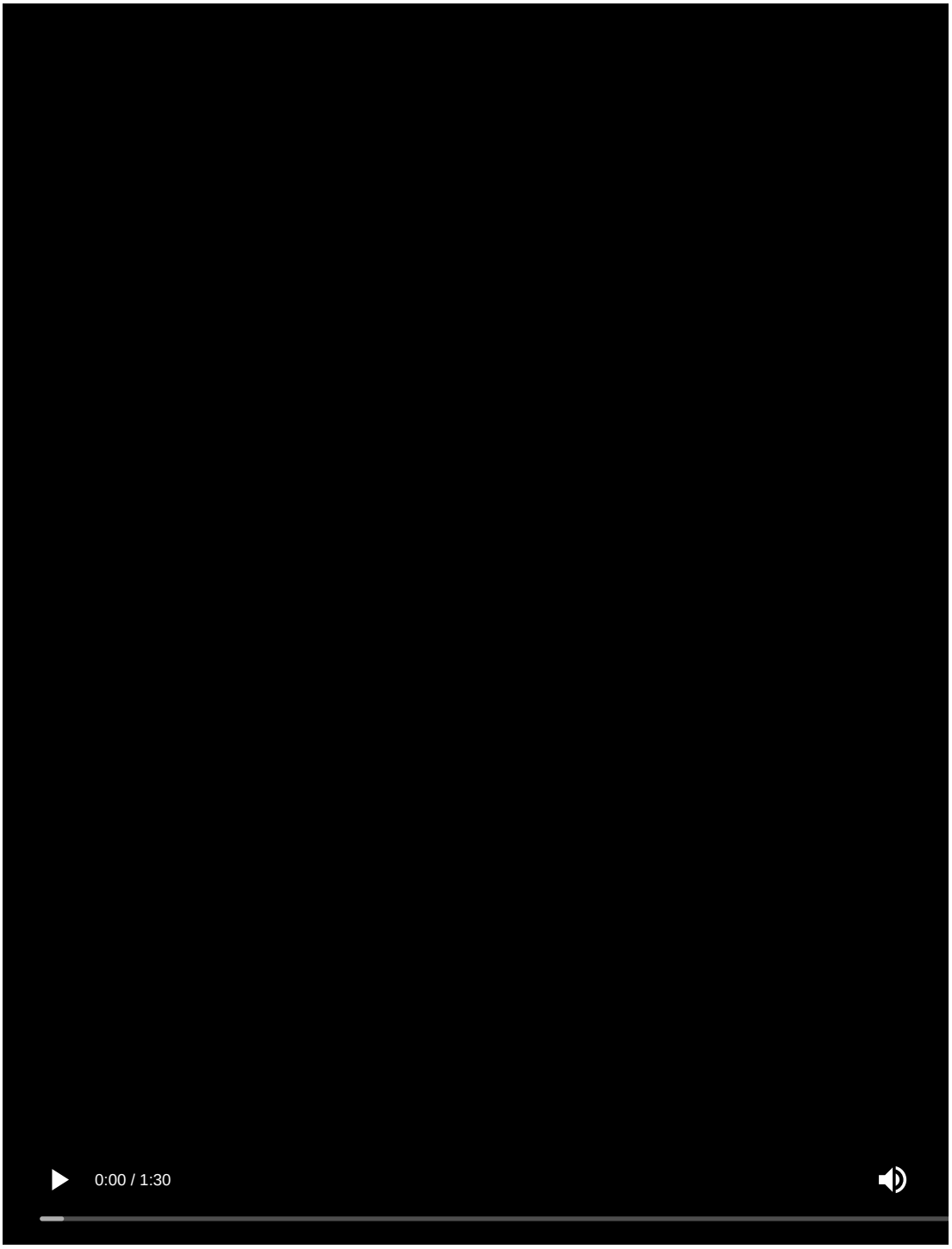
The identifier is a dynamic number that is compared with the action list's identifier for integrity purposes. None of the three actions above are Trusted Installer enforced. This allowed me to update the system with a custom, downgrading action list. All of the integrity verifications were bypassed, since the action list is assumed verified because it is created post-verification.

As a result, there is no need to perform the malicious Trusted-Installer elevation. Instead, Windows Updates did all of the work for me! I was able to achieve a complete Windows Update takeover with a downgrade attack that is:

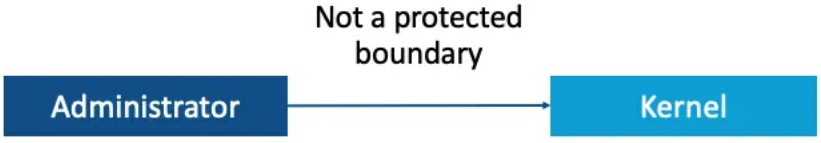
- **Fully undetectable.** Since it was performed in a legitimate way, no malicious activity is detected.
- **Invisible.** Because I technically "updated" the system, it will appear fully up-to-date.
- **Persistent.** I discovered that the action list parser *poqexec.exe* is not digitally signed. As a result, I was able to patch it to install empty updates, meaning any newly available updates will be falsely installed.
- **Irreversible.** I also discovered that the integrity and repair utility SFC.exe is not digitally signed. I was able to patch it as well, meaning it will no longer detect any corruptions. I also found *DISM.exe*, but it detects corruptions in the component store, which there is no reason to modify.

Demo

To see these capabilities in action, the following demo shows how I was able to downgrade a kernel driver named *AFD.sys* to an old and vulnerable version. Next, it shows how I exploited that same kernel driver to achieve kernel code execution.



So far, I've developed a perfect downgrade ability and gained kernel code execution. Since my starting point was Administrator, this was an Administrator-to kernel elevation.



Administrator-to-kernel is not considered a security boundary on Windows. Although not a boundary, Administrator-to-kernel is still a threat and a serious one. Nowadays, many users are still running as Administrators, and running as Administrator is also default on Windows. As a result, many users could easily compromise the kernel, and this is an issue that Microsoft aimed to solve.

Microsoft's solution to this was to deprivilege the kernel to make kernel access less valuable. This de-privileging was done by introducing a feature called virtualization-based security (VBS).

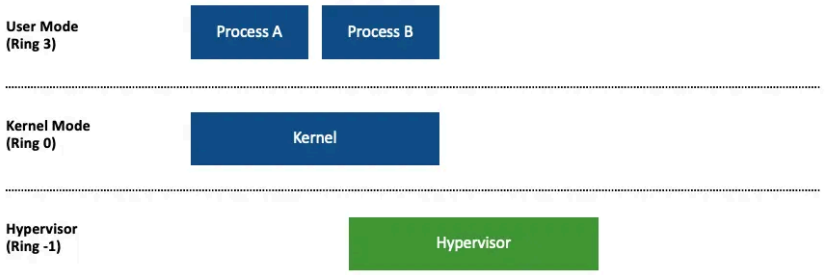
Windows VBS

VBS provides a secure and isolated virtual environment—powered by the Hyper-V hypervisor. The reason that VBS was created is because the kernel is assumed compromised, and there was a need for a secure place to implement security features and store secret keys.

VBS features include Credential Guard, HVCI, System Guard Secure Launch, Shielded VMs, and more security features that really improved the security of Windows.

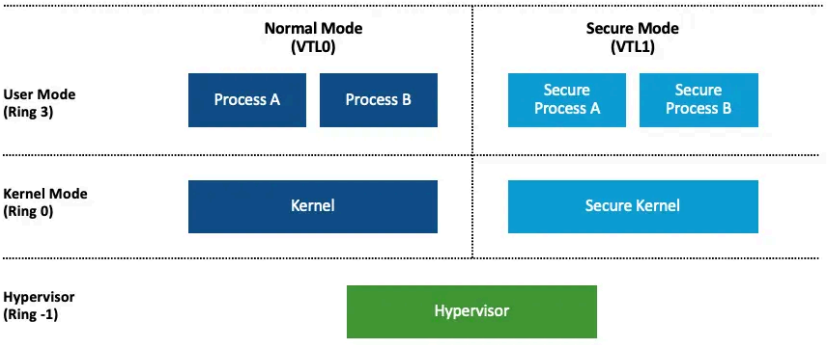
VBS Architecture

Before VBS, the Windows architecture included User Mode in ring 3, the kernel in ring 0, and the hypervisor in ring -1.



With VBS, the hypervisor introduces virtual trust levels (VTLs), which are essentially like virtual machines. The higher the VTL, the more privileged it is. Lower VTLs should not be able to access memory or compromise higher VTLs. Currently on Windows, only two VTLs exist.

VTL0 is known as Normal Mode and contains the original OS. VTL1 is known as Secure Mode, and contains security features and mitigations, as well as key isolation technologies.

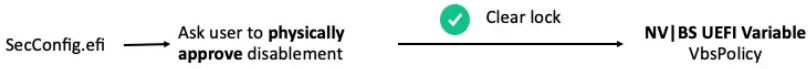


VBS Remote Disablement Protection

As I explored VBS security features and mitigations, I wondered if VBS features could be easily turned off. Allowing VBS configuration modifications by Administrators would be ineffective, since attackers would just turn VBS off instead of dealing with it.

VBS protects against disablements by implementing a feature called UEFI lock. The lock is a UEFI Boot Service variable that holds the VBS configuration. If VBS is configured with a lock, the lock is the source of configuration instead of the Windows Registry. So, changing the registry configuration will have no effect. The UEFI variable is the source of truth. And of course the Boot Service variable can only be accessed during boot, not in runtime from the OS. This is a very simple and reliable protection.

If the user really wants to disable VBS, they must load an EFI application signed by Microsoft. This EFI would then ask the user during boot to physically approve the disablement—only then is VBS disabled. The key element here is that this is performed during boot, and the assumption is that only signed code is allowed to run because of Secure Boot. The only signed code that touches the variable requires physical approval. So without physical access to the target machine, it should not be possible to disable VBS.



Bypassing VBS UEFI Lock

Now that I had the ability to control any file on the OS, I wondered what would happen if I tried to replace the Secure Kernel or the hypervisor powering VBS with invalid files that, for example, are not digitally signed and are attacker-controlled. I assumed the machine would boot into the recovery environment, since invalid files are a sign of compromise.

Instead, I was extremely surprised to find that the OS loader booted normally, abandoning VBS if it failed to validate one of VBS's files. This process allowed me to disable VBS, discovering what I believe to be the first bypass of VBS's UEFI lock!

By combining the downgrade flow I developed and the UEFI lock bypass, I was able to achieve credential extraction against the most restrictive settings:

- PPL was enabled for LSASS with UEFI lock, protecting LSASS dumps.
- Credential Guard was enabled with UEFI lock, protecting secrets in VTL1.
- Windows Defender was up and running.

I bypassed these settings by first reverting the PPLFault patch to allow PPL bypass. Second, I disabled Credential Guard, bypassing the UEFI locks by invalidating the Secure Kernel. Finally, I made Windows Defender non-functional by invalidating MsMpEng, the main Defender executable.

It is important to understand that Credential Guard and PPL for LSASS are complimentary. If only Credential Guard is bypassed, LSASS cannot be dumped.

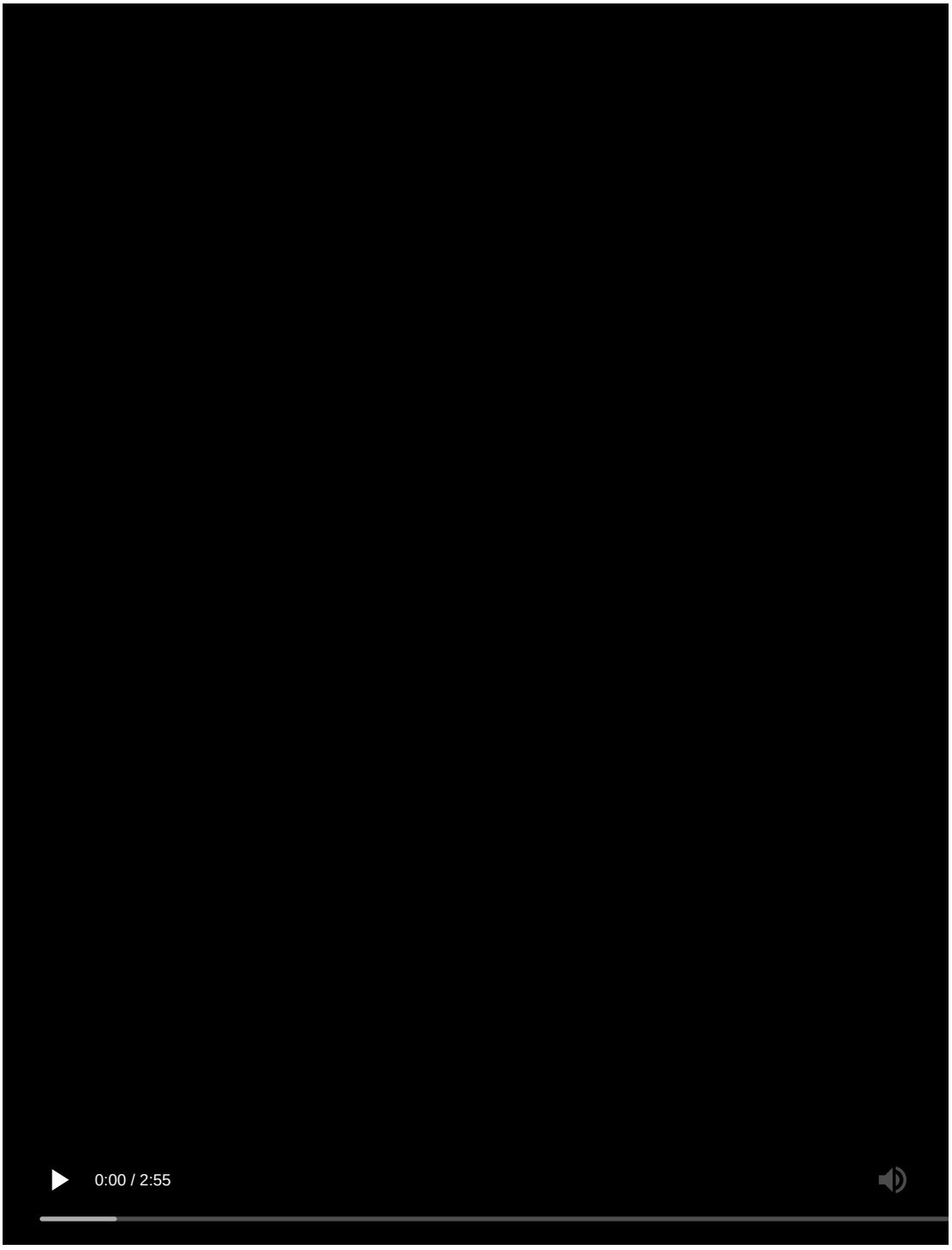
```
C:\Users\Alon\Desktop\Excluded>PPLFault.exe -v 760 lsass.dmp
[+] No cleanup necessary.  Backup does not exist.
[+] GetShellcode: 528 bytes of shellcode written over DLL entrypoint
[+] Benign: C:\Windows\System32\EventAggregation.dll.bak
[+] Payload: C:\PPLFaultTemp\PPLFaultPayload.dll
[+] Placeholder: C:\PPLFaultTemp\EventAggregationPH.dll
[+] Acquired exclusive oplock to file: C:\Windows\System32\devobj.dll
[+] Ready.  Spawning WinTcb.
[+] SpawnPPL: Waiting for child process to finish.
[+] FetchDataCallback called.
[+] Hydrating 90112 bytes at offset 0
[+] Switching to payload
[+] Emptying system working set
[+] Working set purged
[+] Give the memory manager a moment to think
[+] Hydrating 90112 PAYLOAD bytes at offset 0
[!] Did not find expected dump file: lsass.dmp
```

And if only PPL is bypassed, credentials are encrypted and not usable.

```
Authentication Id : 0 ; 466417 (00000000:00071df1)
Session           : Interactive from 1
User Name         : fcastle
Domain            : DEKU
Logon Server      : DEKU-DC
Logon Time        : 7/5/2024 12:13:03 PM
SID               : S-1-5-21-1272771627-2707523857-1367165163-1103
        msv :
         [00000003] Primary
         * Username : fcastle
         * Domain   : DEKU
           * LSA Isolated Data: NtlmHash
             KdfContext: db0f131340c395ea790ee6896f146184b1c08a6dd3b7f56a874e4383560905a4
             Tag       : 197ed55ab8ca142dd067c9f8ccc38e93
             AuthData  : 010000000000000000000000000000000100000034000004e746c6d48617368
             Encrypted : 9c601dc3c53f49df535bee643ddf5fcf6e051cb703e6ddff8e32ba19f452d2852634daee854ef3196c32ccc661958d2ffa296816
         * DPAPI    : 732646f2969ff7857b666e68ba3dca9b
        tspkg :
        wdigest :       KO
        kerberos :
         * Username : fcastle
         * Domain   : DEKU.LOCAL
         * Password : (null)
        ssp :
        credman :
        cloudap :
```

Demo

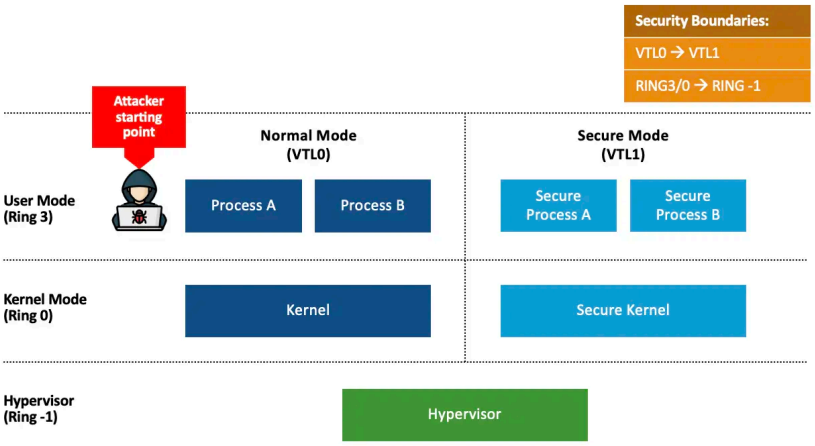
To see these capabilities in action, the following demo shows how I was able to bypass Credential Guard, PPL, and Windows Defender all at once by utilizing my downgrade flow and the UEFI lock bypass.



VBS Security Boundaries

Now that I bypassed the UEFI locks and disabled VBS, it was time to investigate real security boundaries of VBS.

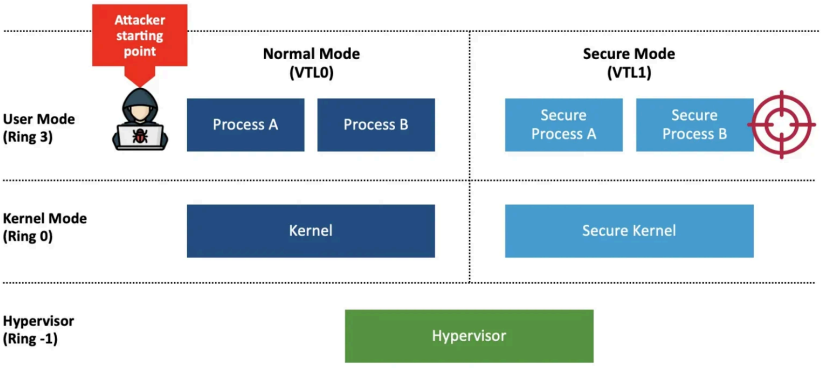
The security boundaries VBS introduces are that any VTL0-to-VTL1 transition is considered an elevation of privilege. No VTL0 code should be able to compromise VTL1 code. Another boundary is that ring 3 to ring -1 or ring 0 to ring -1 (i.e., any user mode or kernel to hypervisor) are also considered an elevation of privilege. Due to the way VBS is implemented, compromising the hypervisor gives control over all VTLs.



When considering VBS downgrades, my main goal was to understand if there was a downgrade mitigation—like versioning checks or a revocation mechanism—in the virtualization stack. If no downgrade mitigation was found, I would attempt to downgrade to vulnerable code. Regardless of how far I could go with the downgrade now, I would consider this a vulnerability. That is because it would make exploitation much easier if a new vulnerability was found and patched in the target component at some point in the future. And even if not immediately exploitable, it is something that needs to be addressed.

Targeting Secure Mode's Isolated User Mode Processes

To begin, I set my sights on Secure Mode's isolated user mode processes. I decided to target the most well-known isolated user mode process: Credential Guard.



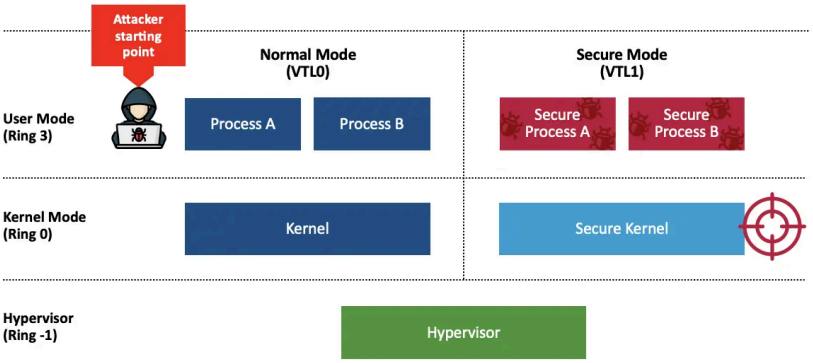
Credential Guard is implemented in Ring3-VTL1 as an isolated user-mode process named *LsaIso.exe*, which stands for Lsa Isolated. With Credential Guard running, secrets are stored in *LsaIso.exe* in VTL1, instead of in the original LSASS process. This is why dumping LSASS is no longer valuable, because LSASS only contains encrypted blobs. The real secrets are not accessible to any VTL0 code.

I targeted the downgrade with CVE-2022-34709, which is a fixed elevation of privilege in Credential Guard. I found both the vulnerable module, *KerbClientShared*, and the vulnerable version 10.0.22000.856. Downgrading the vulnerable module using the Windows Update takeover worked on a fully patched machine!

This allowed me to escalate privileges from Ring3-VTL0 to Ring3-VTL1, compromising Credential Guard and the secrets it protects. **There was no downgrade detection on VTL1 isolated user mode!**

Targeting Secure Mode's Kernel

Isolated User Mode privileges are great, but I wanted to gain higher privileges like those found in Secure Mode's kernel.



The kernel of Secure Mode is called Secure Kernel, and it is a minimal kernel that only implements security features such as HVCI, HyperGuard, and more.

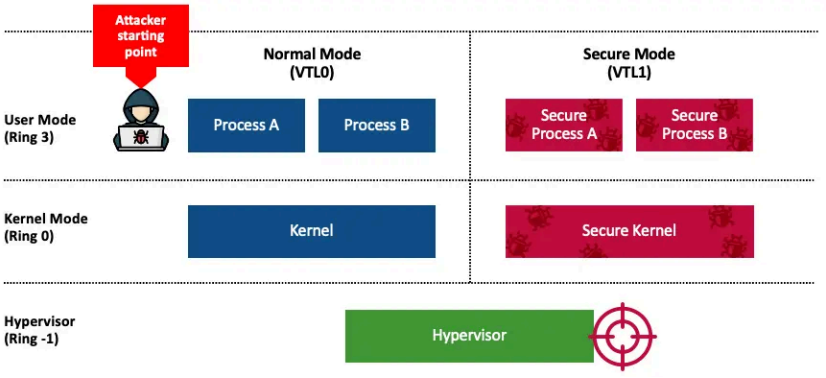
I targeted the downgrade with CVE-2021-27090, which is a fixed elevation of privilege in Secure Kernel. The vulnerable module here could only be Secure Kernel (*SecureKernel.exe*) and the vulnerable version 10.0.19041.207.

However, in this case, downgrading only Secure Kernel was not enough. I also had to downgrade dependency modules—like *SKCI.dll* and *CI.dll*—and then the Windows Update takeover worked on a fully patched machine!

This allowed me to escalate privileges from Ring3-VTL0 to Ring0-VTL1, compromising the entire VTL and any of its mitigations, including HVCI, HyperGuard, and more. **There was no downgrade detection on VTL1 Secure Kernel either!**

Targeting Hyper-V's Hypervisor

Secure kernel privilege is awesome, but I wanted to set my sights even higher. So, I moved on to target the most privileged entity of VBS: the Hyper-V hypervisor.

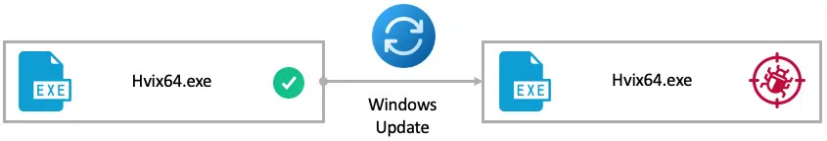


The hypervisor is *Hvix64.exe* on Intel systems and *Hvax64.exe* on AMD systems. The hypervisor is a standalone micro-kernel, which makes it a valuable target for downgrade.

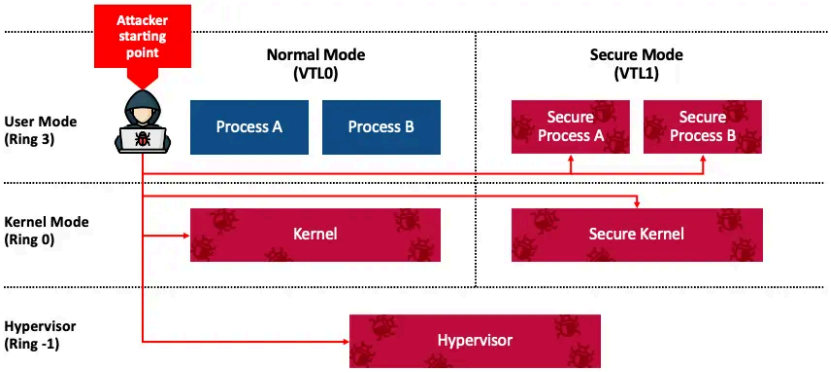
Unlike the isolated user mode process and the secure kernel, it was more challenging to target the hypervisor downgrade with a specific vulnerability. While there were many vulnerabilities titled Hyper-V Elevation of Privilege that had been fixed over the last two years, Hyper-V is a large component. In relevant CVE descriptions, Microsoft doesn't share which component in the Hyper-V stack is vulnerable, so it could be user mode elevated process, kernel driver, or the hypervisor itself, which is the component I am targeting.

So, I decided to downgrade Hyper-V to a version from two years ago, which seemed like enough time that I could be confident there was at least one hypervisor elevation-of-privilege vulnerability that would be unfixed in the version I selected.

Again, I was able to successfully downgrade the hypervisor and the hypervisor loader using the Windows Update takeover!

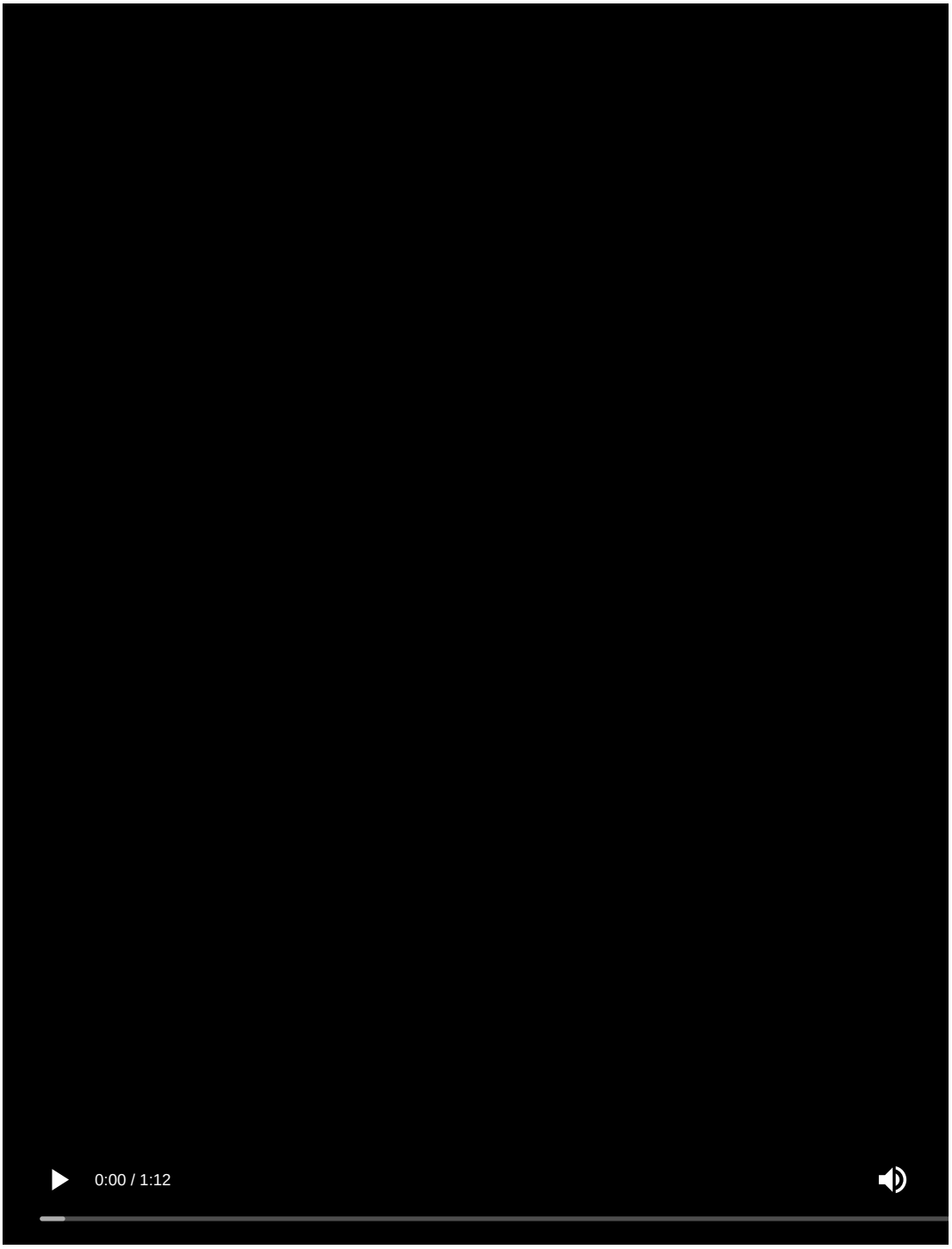


This allowed me to escalate privileges from Ring3-VTL0 to Ring -1, potentially compromising the entire virtualization stack. **I found no downgrade mitigation in any component of the virtualization stack—the entire stack was vulnerable to downgrades.**



Demo

To see these capabilities in action, the following demo shows how I was able to utilize my downgrade flow to successfully downgrade the hypervisor.



Vendor Response

When it comes to our original research, SafeBreach is deeply committed to responsible disclosure. In line with that commitment, we notified Microsoft of our research findings in February 2024. Microsoft issued two CVEs—CVE-2024-21302 and CVE-2024-38202—and sent the following official response:

"We appreciate the work of SafeBreach in identifying and responsibly reporting this vulnerability through a coordinated vulnerability disclosure. We are actively developing mitigations to protect against these risks while following an extensive process involving a thorough investigation, update development across all affected versions, and compatibility testing, to ensure maximized customer protection with minimized operational disruption."

Microsoft provided additional guidance via *Security Update Guide ADV24216903* and has asked that we direct anyone with further questions about their response plan to reach out to the Microsoft communications team directly at rapidresponse@we-worldwide.com.

Conclusion

This research set out to explore the Windows Update process to identify how it might be manipulated to enable a downgrade attack. I was able to show how it was possible to make a fully patched Windows machine susceptible to thousands of past vulnerabilities, turning fixed vulnerabilities into zero-days and making the term “fully patched” meaningless on any Windows machine in the world. We believe the implications are significant not only to Microsoft Windows, which is the world’s most widely used desktop OS, but also to other OS vendors that may potentially be susceptible to downgrade attacks. To help mitigate the potential impact of the vulnerabilities identified by this research, we:

- Responsibly disclosed our research findings to Microsoft in February 2024, as noted above.
- Shared my research openly with the broader security community here and at my Black Hat USA 2024 and DEF CON 32 (2024) presentations to enable the organizations and end-users leveraging the Windows OS to better understand the risks associated with these vulnerabilities.
- Provided a research repository that includes tools that enable the verification and exploitation of these vulnerabilities to serve as a basis for further research and development.
- Added the following attack content to the SafeBreach platform that enables our customers to validate their environment against the vulnerabilities and techniques outlined in this research to significantly mitigate their risk. These attacks allow an attacker to potentially take over the Windows Update process, downgrade Windows to an older version, and make it vulnerable to any known one-day vulnerability.
    - #10341 – Windows Downdate – Windows update takeover
    - #10342 – Windows Downdate – TrustedInstaller elevation

For more in-depth information about this research, please:

- Contact your customer success representative if you are a current SafeBreach customer
- Schedule a one-on-one discussion with a SafeBreach expert
- Contact Kesselring PR for media inquiries

Credits

I would also like to give credit to the talented individuals below for their work:

- James Forsahw (@tiraniddo) – CVE-2022-34709
- Saar Amar (@AmarSaar) – CVE-2021-27090
- Gabriel Landau (@GabrielLandau) – PPLFault
- Valentina Palmiotti (@chompie1337), Ruben Boonen (@FuzzySec) – CVE-2023-21768 Exploit
- Benjamin Delphi (@gentilkiwi) – Mimikatz

About Our Researcher

Alon Leviev (@_0xDeku) is self-taught security researcher with a diverse background. Alon started his professional career as a blue team operator, where he focused on the defensive side of cyber security. As his passion grew towards research, Alon joined SafeBreach as a security researcher. His main focus included operating system internals, reverse engineering, and vulnerability research. Alon has spoken at various security conferences like Black Hat Europe 2023, CanSecWest 2024, and CONFidence 2024. Before joining the cyber security field, Alon was a professional Brazilian jiu-jitsu athlete, where he won several world and European titles.
