BlueHat IL

Recovery Reimagined: Attacking and Securing Windows Recovery

Who Are We?

Security Testing & Offensive Research at Microsoft (STORM)

Alon Leviev (@alon_leviev)
Senior Security Researcher @ Microsoft

Netanel Ben Simon (@NetanelBenSimon)
Senior Security Researcher @ Microsoft

Microsoft

STORM

BlueHat IL

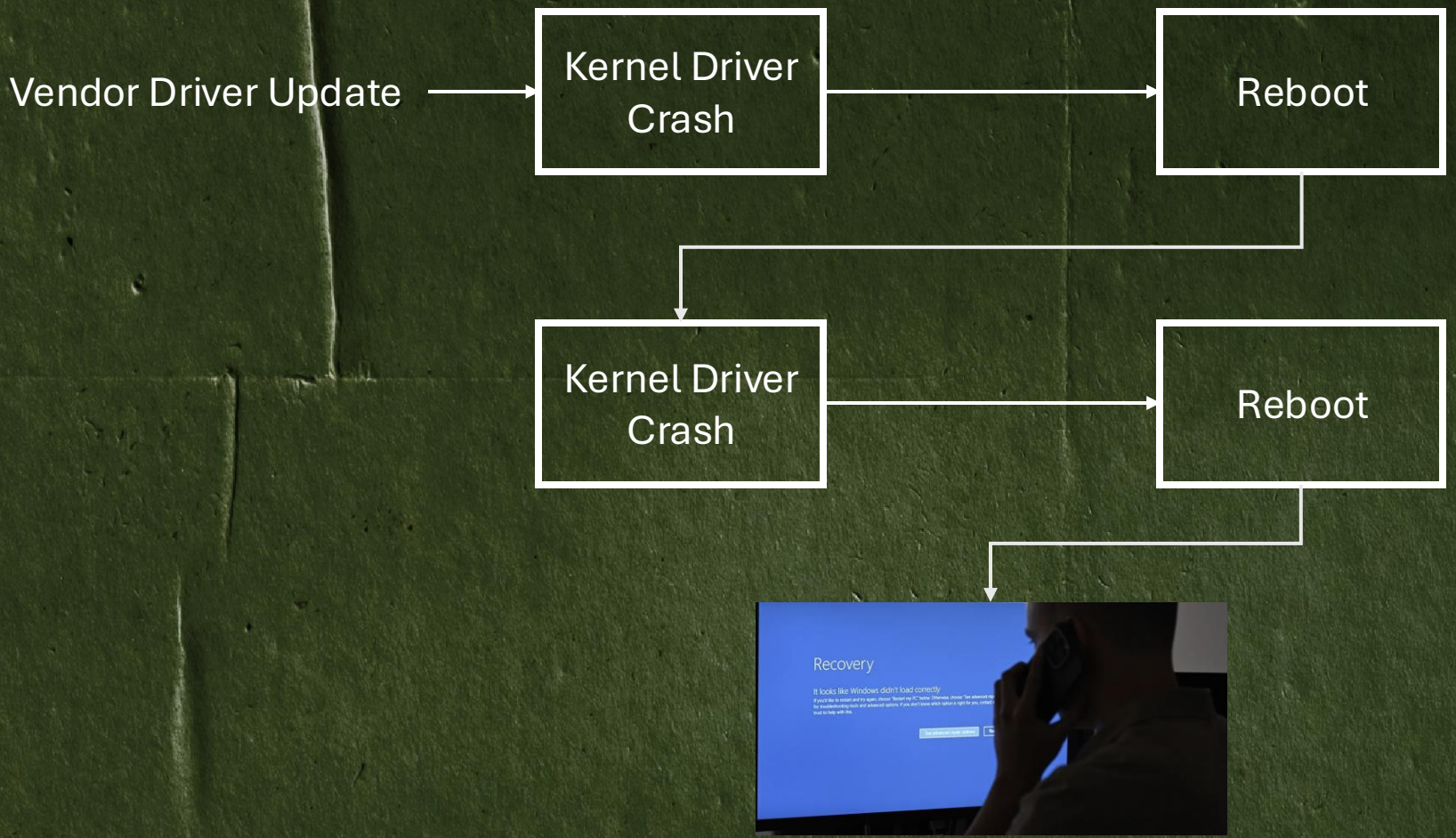
Agenda

- Research Background
- WinRE Overview
- The Vulnerabilities
- WinRE Research Helpers
- Closing Remarks

July 19th, 2024 – The Global Outage

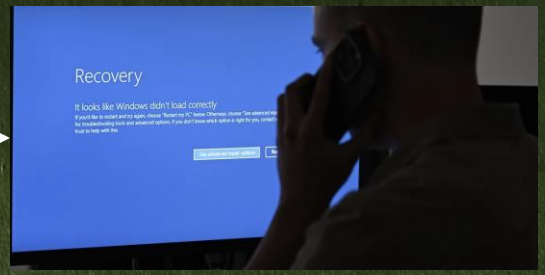
BlueHat IL

What Happened?



The Fix Procedure

Manual and physical recovery needed

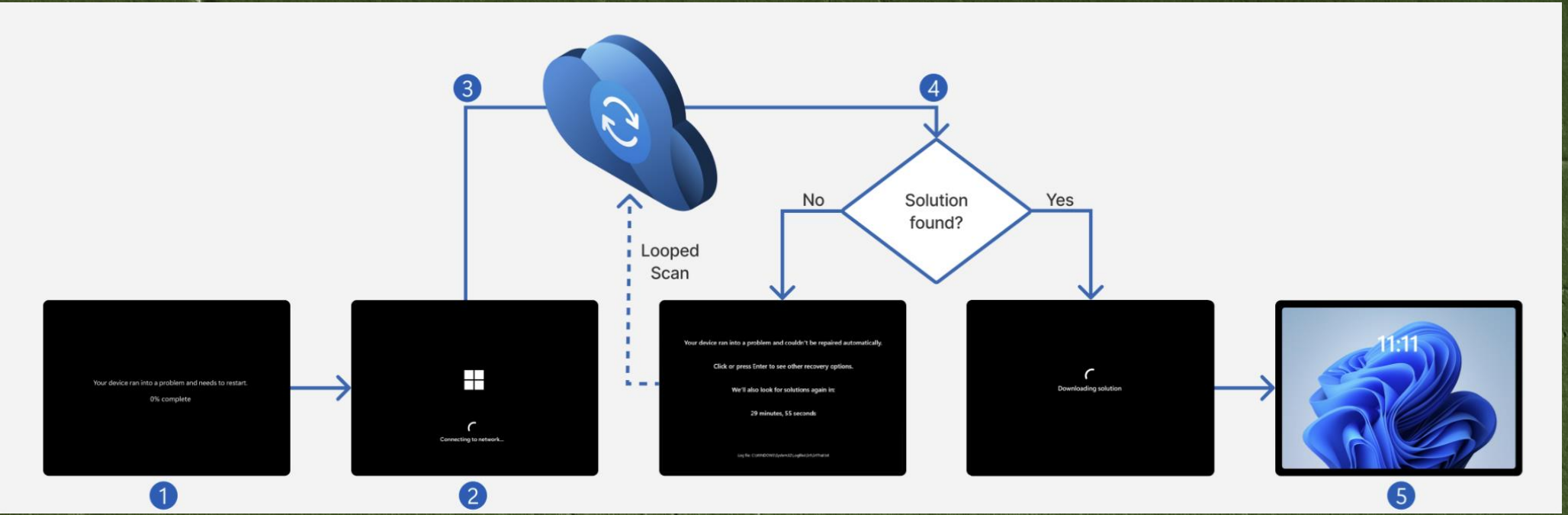


The Windows Resiliency Initiative (WRI)

- A Microsoft initiative to make Windows more resilient
- Builds stronger recovery and resiliency capabilities

WRI Feature – Quick Machine Recovery (QMR)

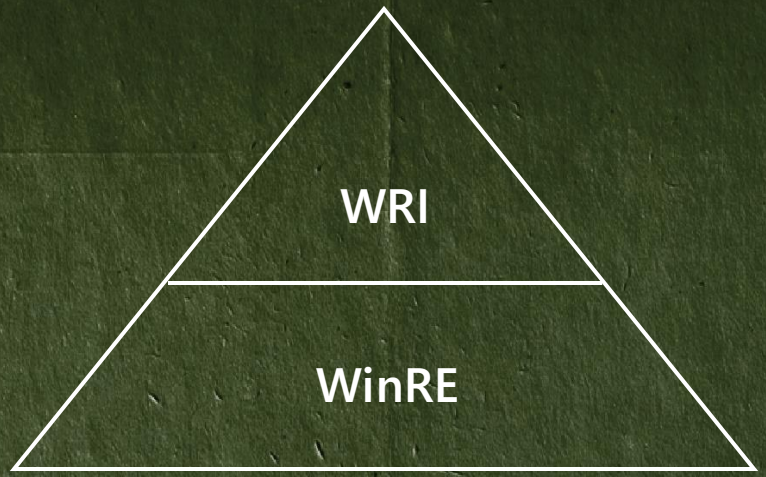
- Automates recovery from critical failures
- Designed to handle large-scale outage scenarios
- On failure, retrieves and applies the required fix from update services



BlueHat IL

WRI Foundation – Windows Recovery

- The foundation – the Windows Recovery Environment (WinRE)

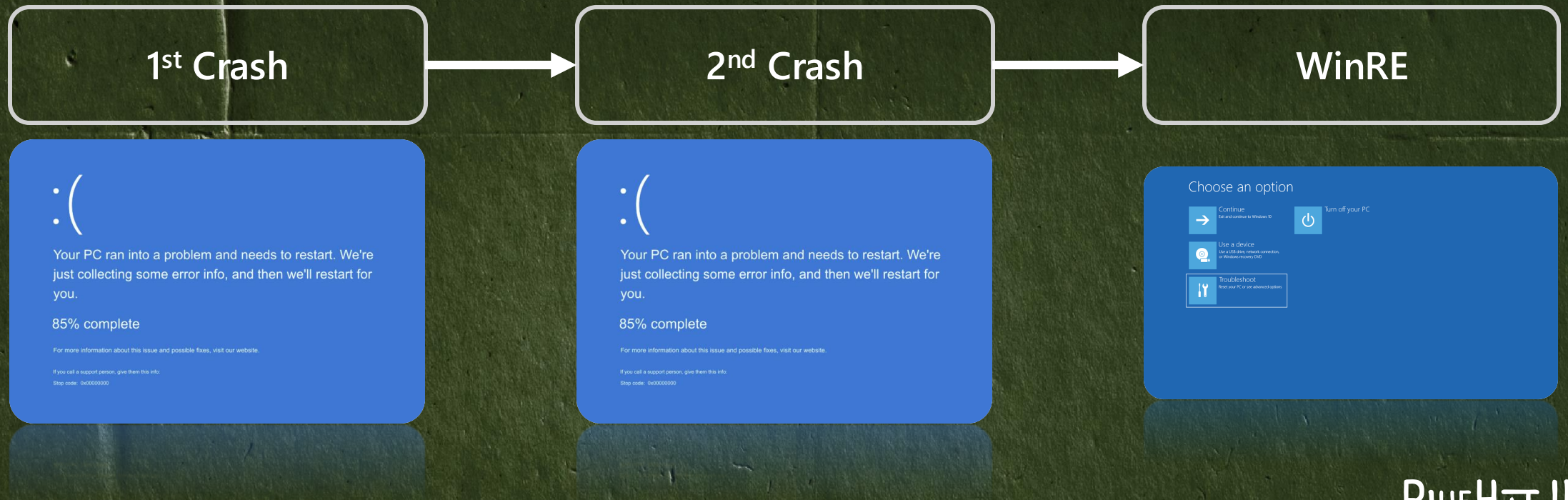


Our Task: Security Reviews

- We were tasked with reviewing both new resiliency features and the platform they rely on:
  - Forward-looking: Review new WRI features design and implementations
  - Backward-looking: Review WinRE design and implementation

WinRE Overview

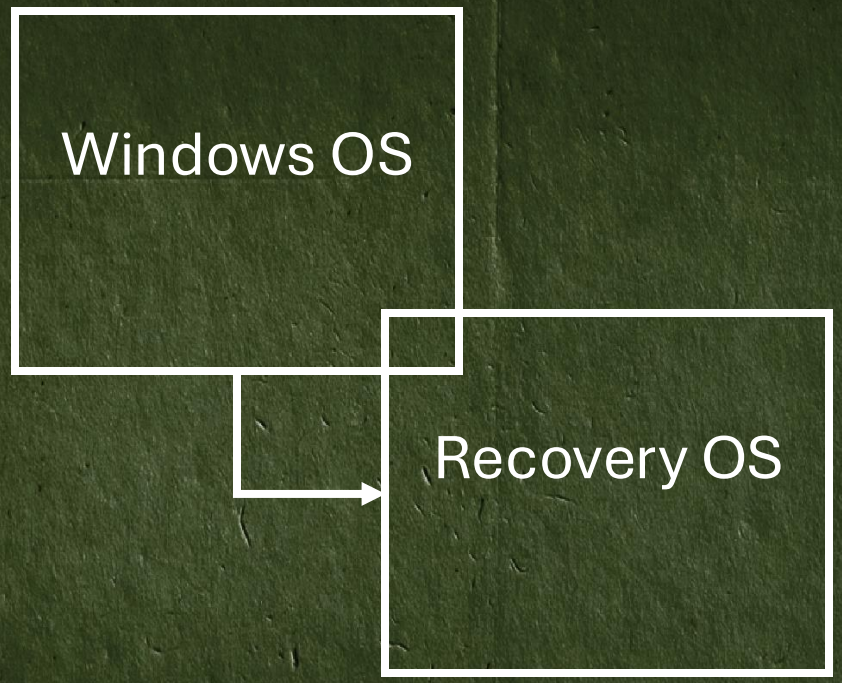
- WinRE is the recovery platform of Windows
- WinRE is designed to recover from critical system issues



BlueHat IL

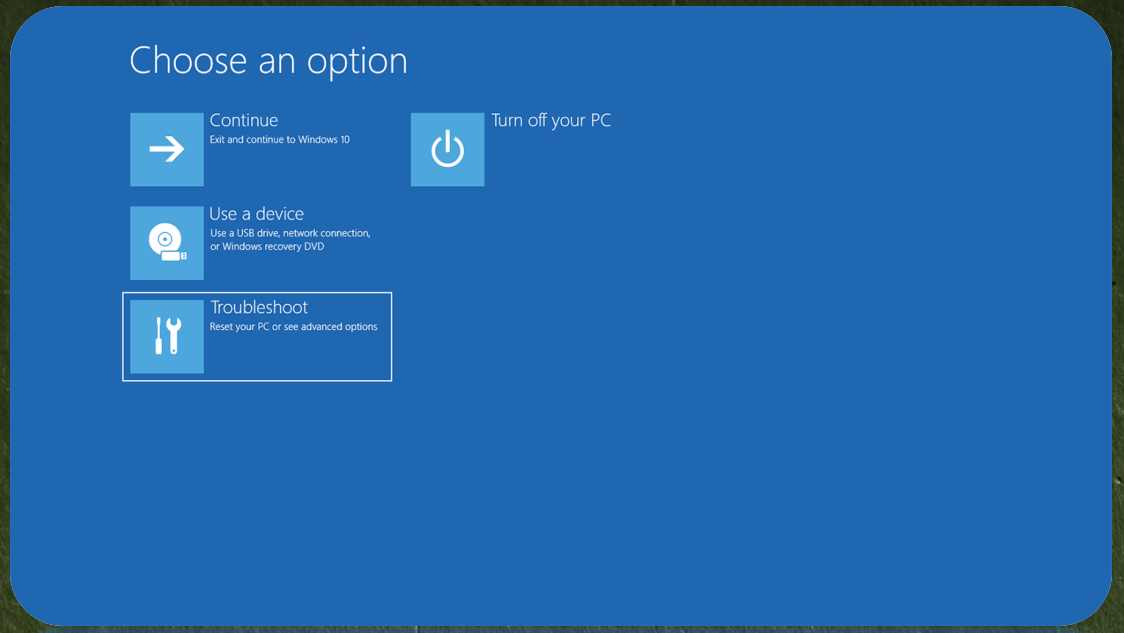
WinRE Architecture – Recovery OS

- WinRE is a lean Windows OS with recovery customizations (aka. Recovery OS)



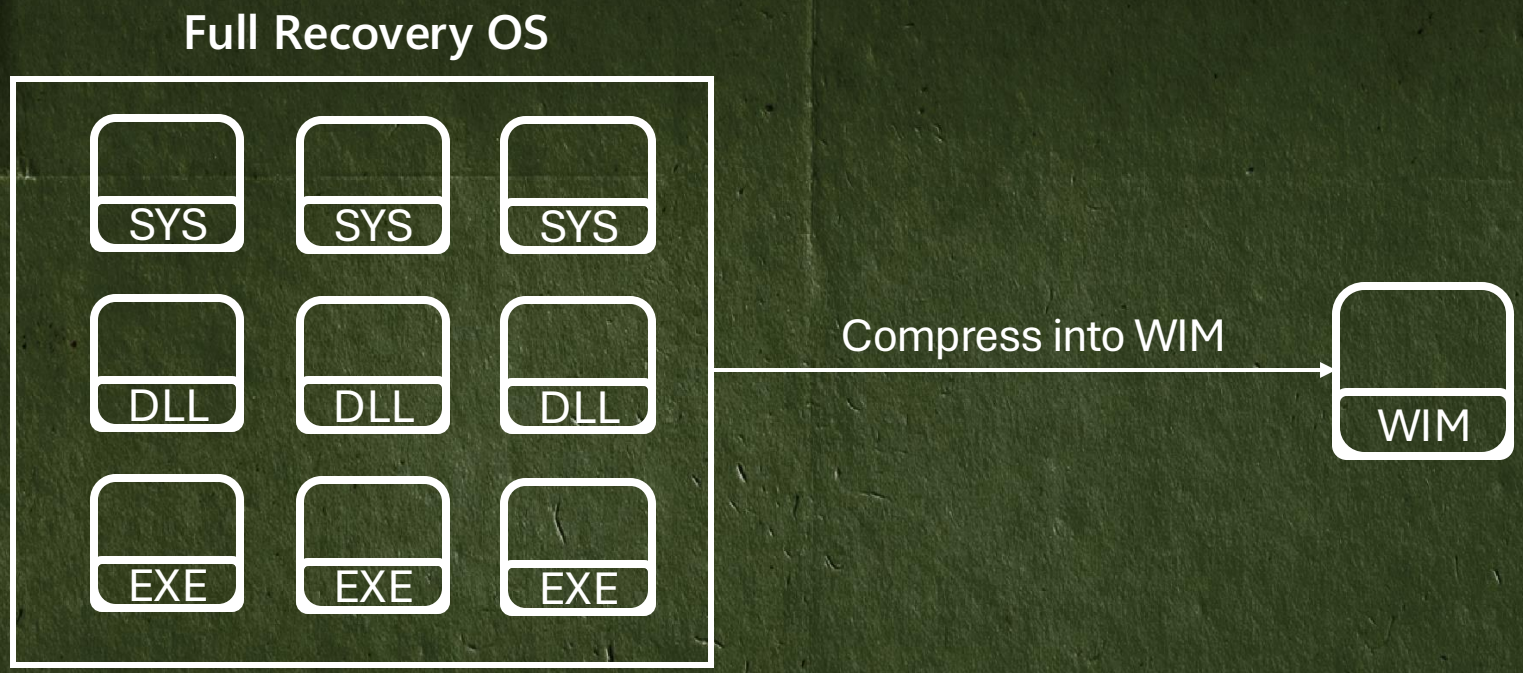
WinRE Architecture – Customizations

- The customizations include Startup Repair, System Reset, System Restore etc.



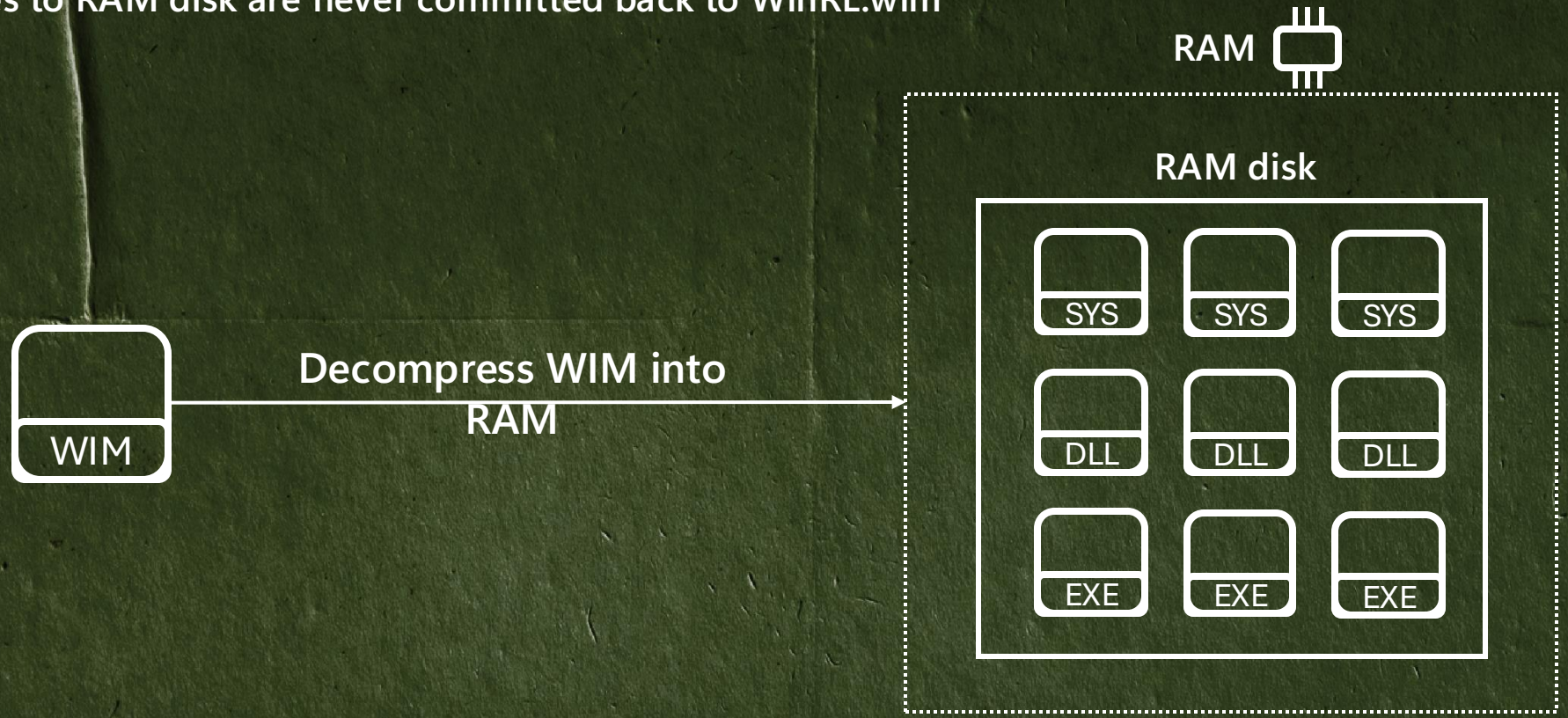
WinRE Architecture – WinRE.wim

- The recovery OS is compressed into a single WIM file – WinRE.wim



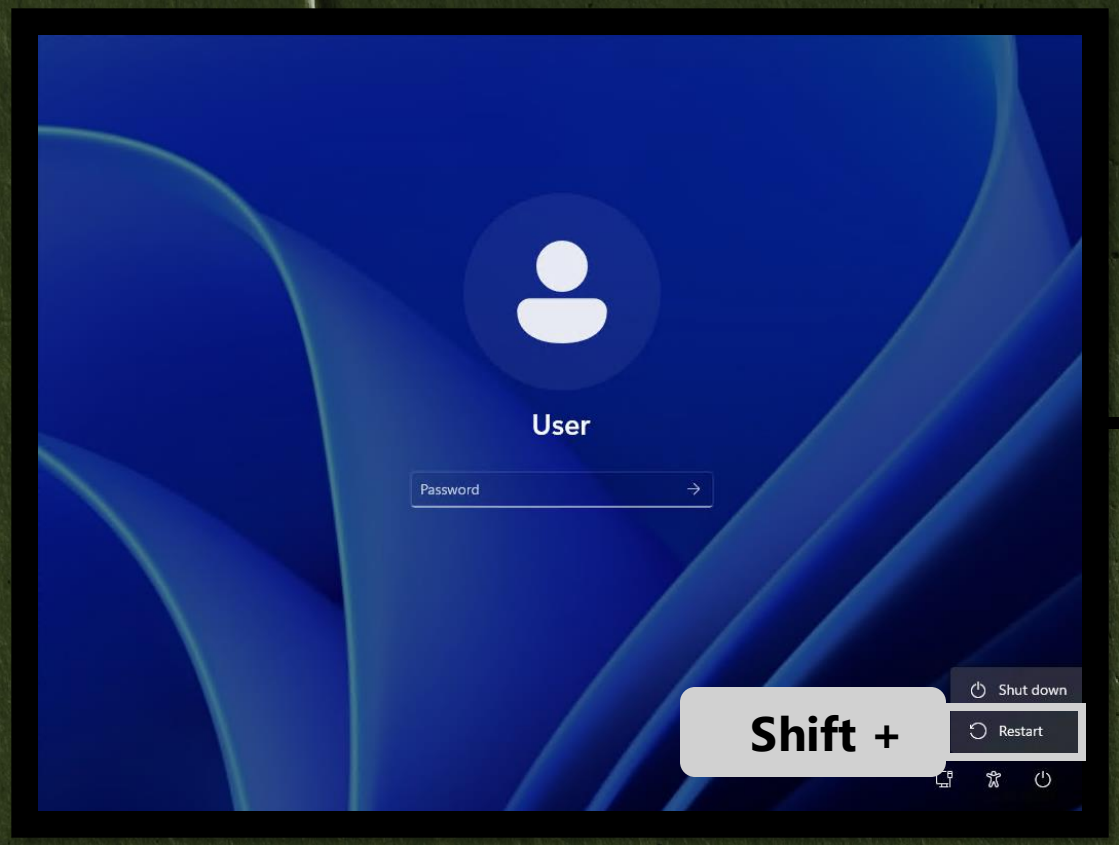
WinRE Architecture – RAM Disk Boot

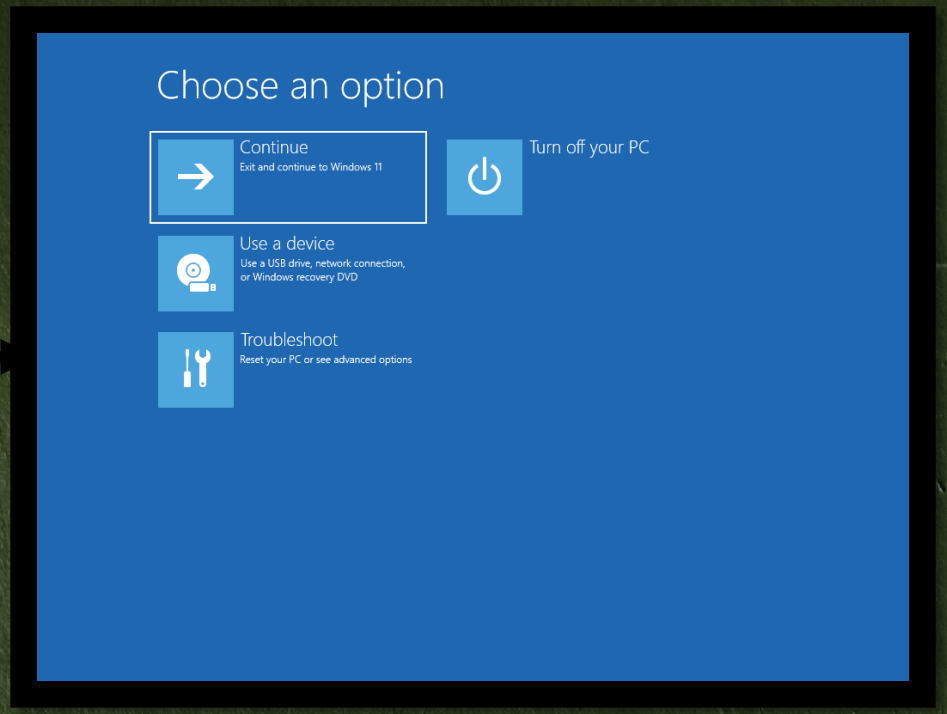
- WinRE.wim is booted from RAM disk
- Changes to RAM disk are never committed back to WinRE.wim



WinRE Interface

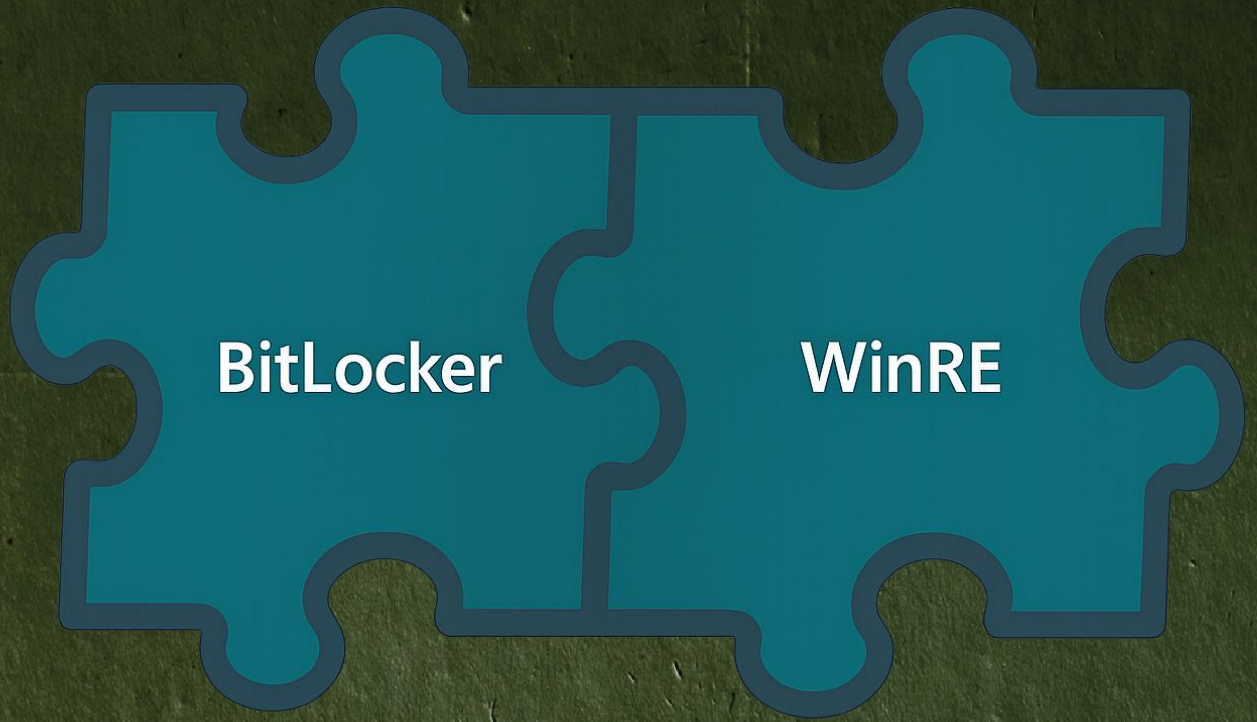
- A physical user can boot into WinRE by pressing Shift + Restart from the logon screen





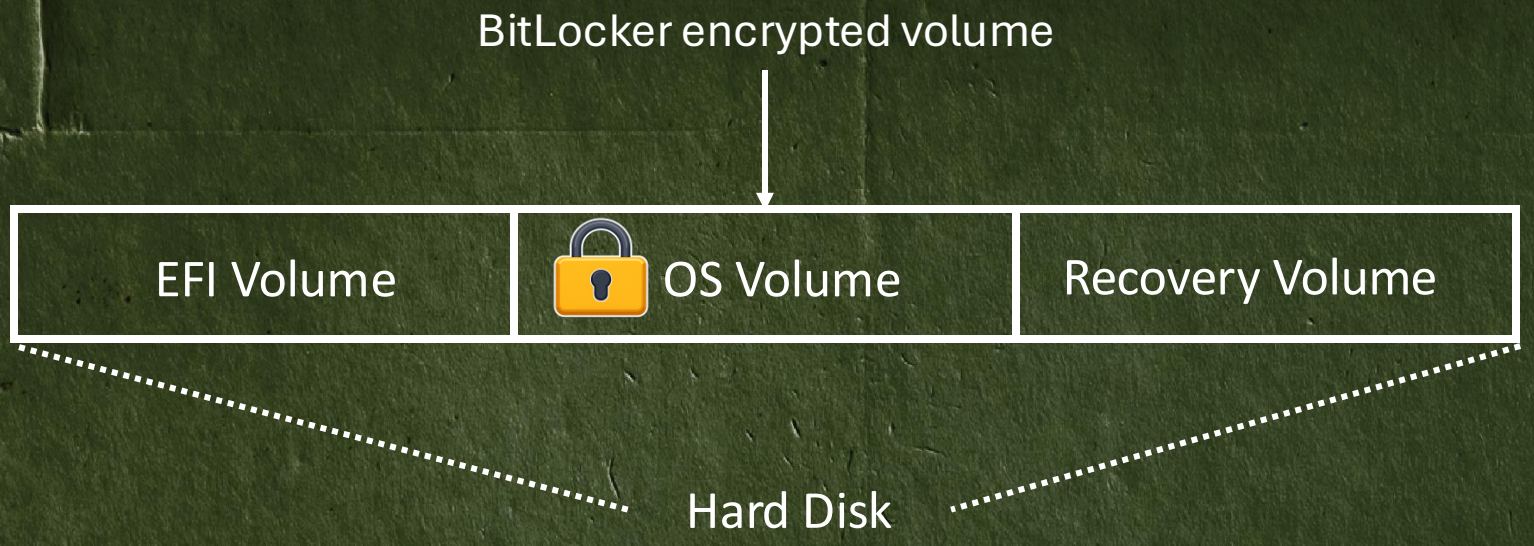
BlueHat IL

WinRE and BitLocker Intersection



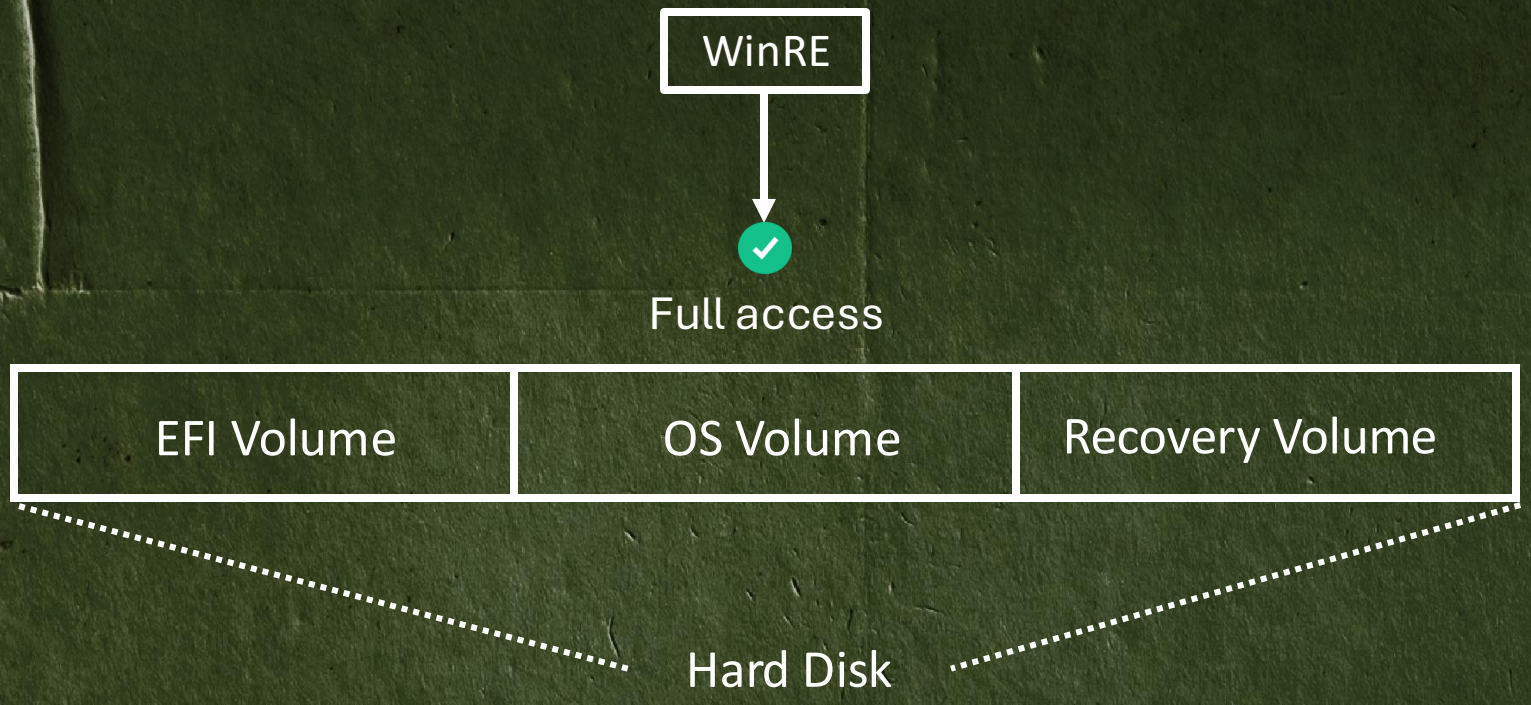
BitLocker Overview

- BitLocker is designed to protect data-at-rest and defend against theft scenarios
- It is a full volume encryption (FVE) technology



WinRE and BitLocker Intersection

- If certain conditions are met – WinRE has full access to the OS volume

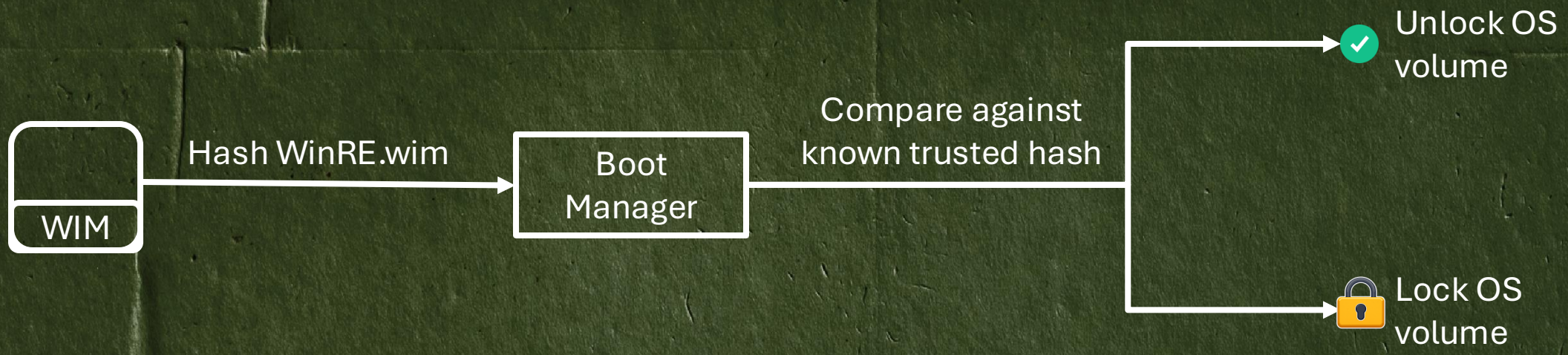


WinRE OS Volume Access Conditions

1. WinRE.wim is trusted
2. Overly permissive recovery tools aren’t selected

Condition #1 – WinRE is Trusted

- Trusted WIM boot compares the WinRE.wim hash against a known trusted hash

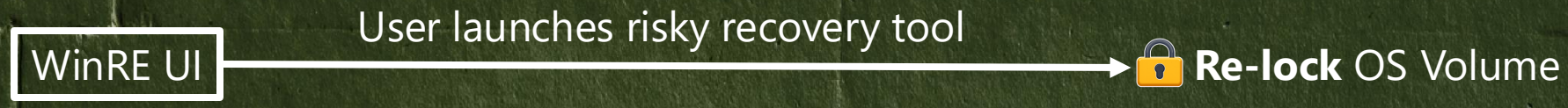


Condition #1 – WinRE is Trusted

- In Unlocked state – WinRE can fully access the OS volume
- In Locked state – WinRE cannot access the OS volume

Condition #2 – Risky Recovery Tools

- If a risky recovery tool is selected from the recovery UI – OS volume is re-locked
- For example: Command Prompt



Research Rules

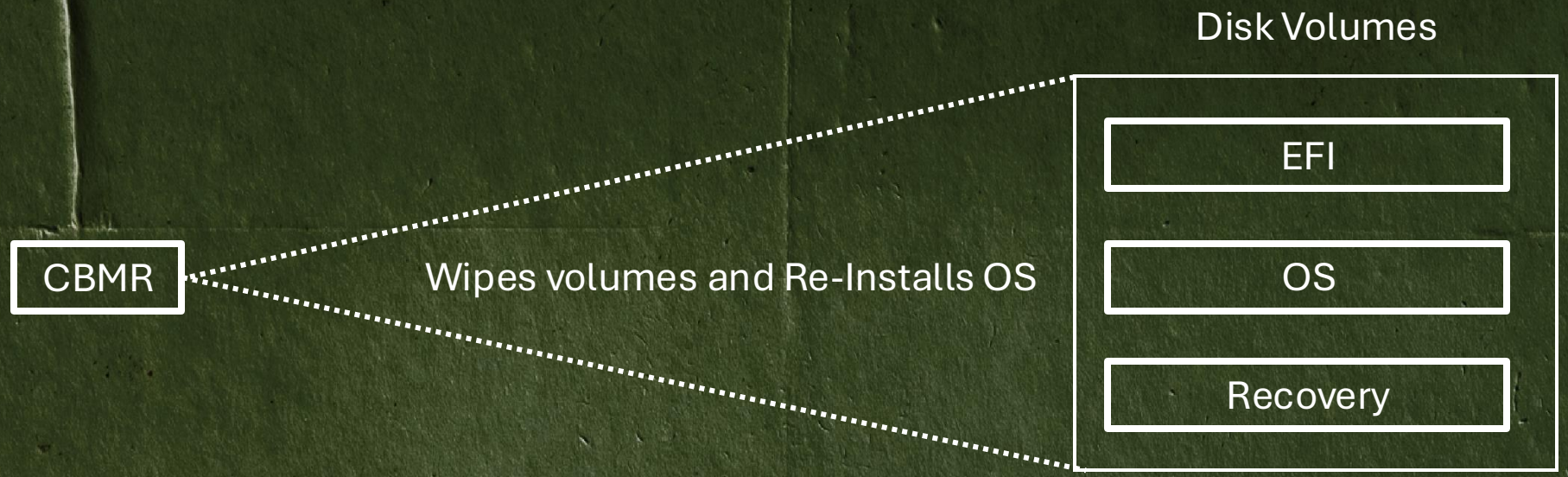
1. Static WinRE.wim image cannot be modified – due to trusted WIM boot
2. Recovery tools that trigger re-lock are out-of-scope

Our Focus Today

- We will dive into:
  -  CVE-2025-55330 – BitLocker Security Feature Bypass
    - CVE-2025-55332 – BitLocker Security Feature Bypass

Cloud Bare Metal Recovery (CBMR) Overview

- Cloud-based bare-metal recovery that reinstalls Windows from scratch via WinRE



CBMR Execution Path – Pseudo Code

- CBMR executes only if:
  - “CBMR” folder exists on the recovery volume
  - There’s no associated OS

```
if (AssociatedOS == nullptr)
{
    AppendPath(RecoveryDrive,
               L"CBMR",
               CbmrFolderPath,
               szCbmrFolderPath);

    if (PathExists(CbmrFolderPath))
    {
        ShouldLaunchCBMR = TRUE;
    }
}

// [REDACTED SNIP]

if (ShouldLaunchCBMR)
{
    LaunchCBMR();
}
```

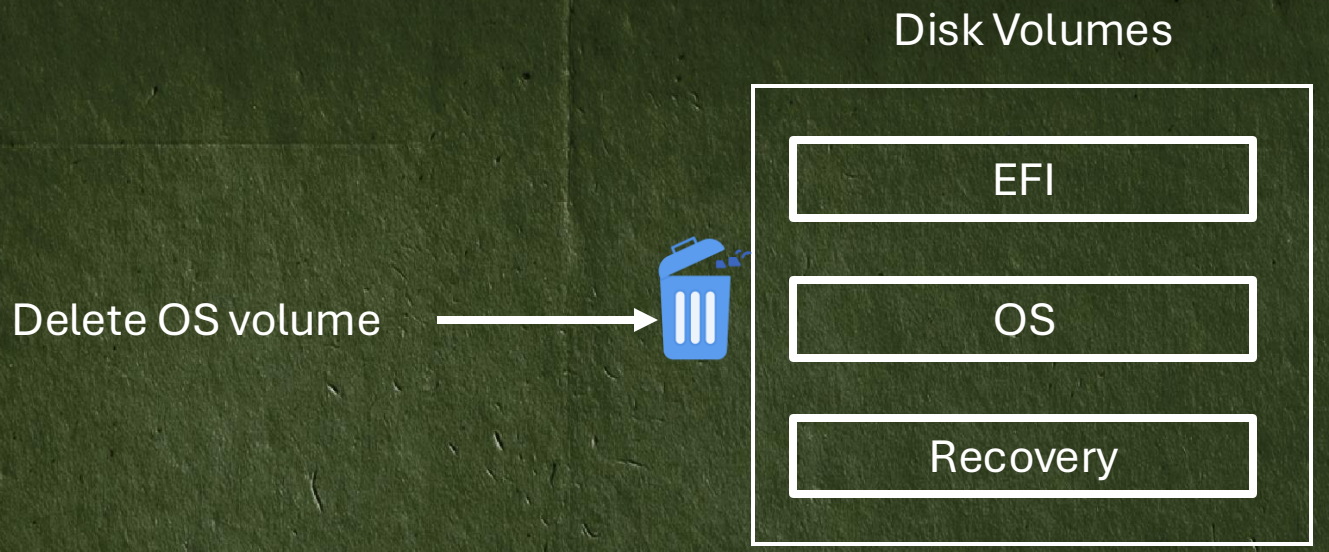
Triggering CBMR

- CBMR executes only if:
  - ✅ “CBMR” folder exists on the recovery volume
  - 👀 There’s no associated OS

```
if (AssociatedOS == nullptr)
{
    AppendPath(RecoveryDrive,
               L"CBMR",
               CbmrFolderPath,
               szCbmrFolderPath);

    if (PathExists(CbmrFolderPath))
    {
        ShouldLaunchCBMR = TRUE;
    }
}

// [REDACTED SNIP]

if (ShouldLaunchCBMR)
{
    LaunchCBMR();
}
```

No OS Volume, No Prize

- If we delete OS volume – we have nothing to attack

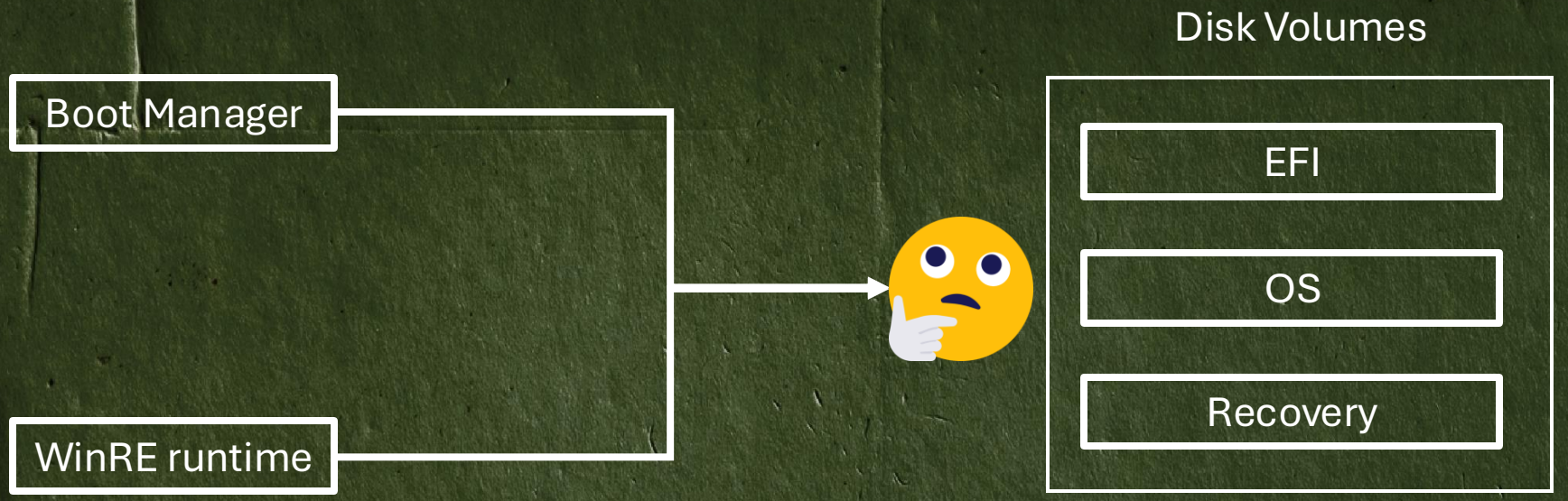


What’s Needed?

1. No OS volume found by WinRE → CBMR triggers
2. OS volume must still exist → target data
3. OS volume must stay unlocked → data accessible

OS Lookups – Happens Twice

- Boot phase – decides what to unlock
- OS phase – decides what to recover
- The lookup logics aren’t shared!



OS Lookups – BCD based

- BCD – Boot Configuration Data
- BCD represents Windows Boot configuration
- BCD is stored on the EFI volume

```
C:\Windows\System32>bcdedit /enum {f227cc65-aa4d-11f0-806c-28a06b34d7de} /v

Windows Boot Loader
-------------------
identifier              {f227cc65-aa4d-11f0-806c-28a06b34d7de}
device                  partition=C:
path                    \WINDOWS\system32\winload.efi
description             Windows 11
locale                  en-US
inherit                 {6efb52bf-1766-41db-a6b3-0ee5eff72bd7}
recoverysequence        {f227cc67-aa4d-11f0-806c-28a06b34d7de}
displaymessageoverride  Recovery
recoveryenabled         Yes
isolatedcontext         Yes
allowedinmemorysettings 0x15000075
osdevice                partition=C:
systemroot              \WINDOWS
resumeobject            {f227cc64-aa4d-11f0-806c-28a06b34d7de}
nx                      OptIn
bootmenupolicy          Standard
hypervisorlaunchtype    Auto
claimedtpmcounter       0x10002
```

Boot Phase Lookup Exclusive Addition

- Boot phase lookups support a dedicated BCD element that specifies the associated OS
- OS phase lookups do not support this BCD element

```
> bcdedit /set {recovery_os_object} custom:0x11000083 partition=TARGET_PARTITION
```

Failing OS Phase Lookups

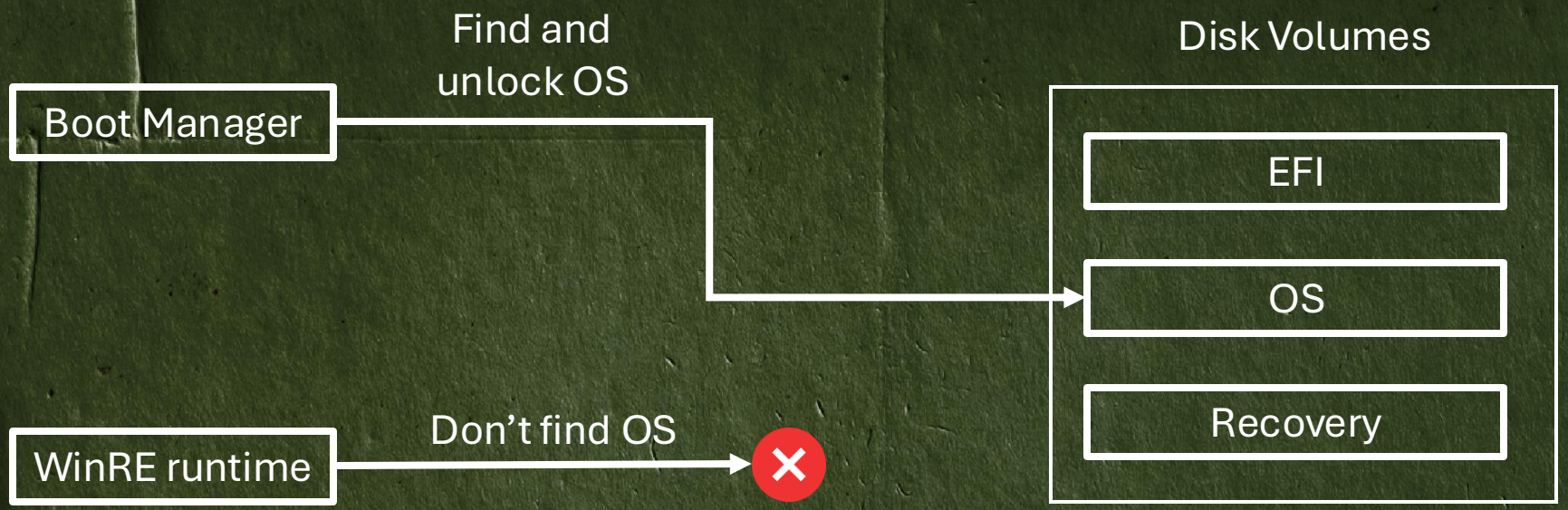
- OS phase lookups heavily rely on the recovery sequence BCD element
- Deleting this element from the target OS BCD object will cause all lookups to fail

```
> bcdedit /deletevalue {target_os_object} recoverysequence
```

The Result of This Setup

```
> bcdedit /set {recovery_os_object} custom:0x11000083 partition=TARGET_PARTITION
```

```
> bcdedit /deletevalue {target_os_object} recoverysequence
```



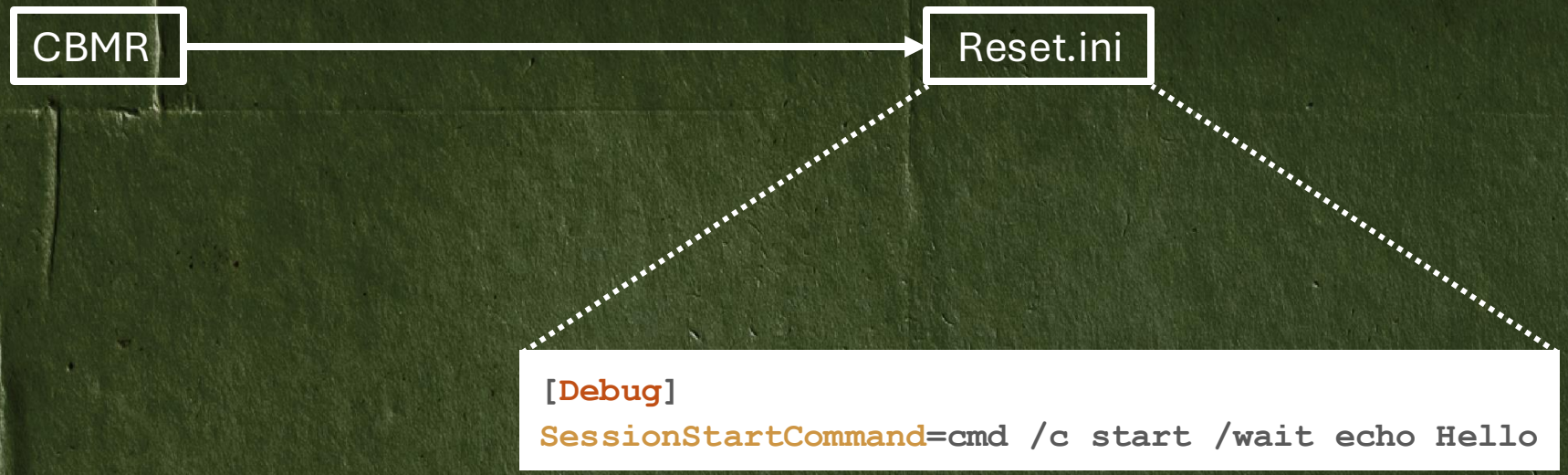
CBMR Execution Path – Pseudo Code

- CBMR executes only if:
  - ✓ “CBMR” folder exists on the recovery volume
  - ✓ There’s no associated OS

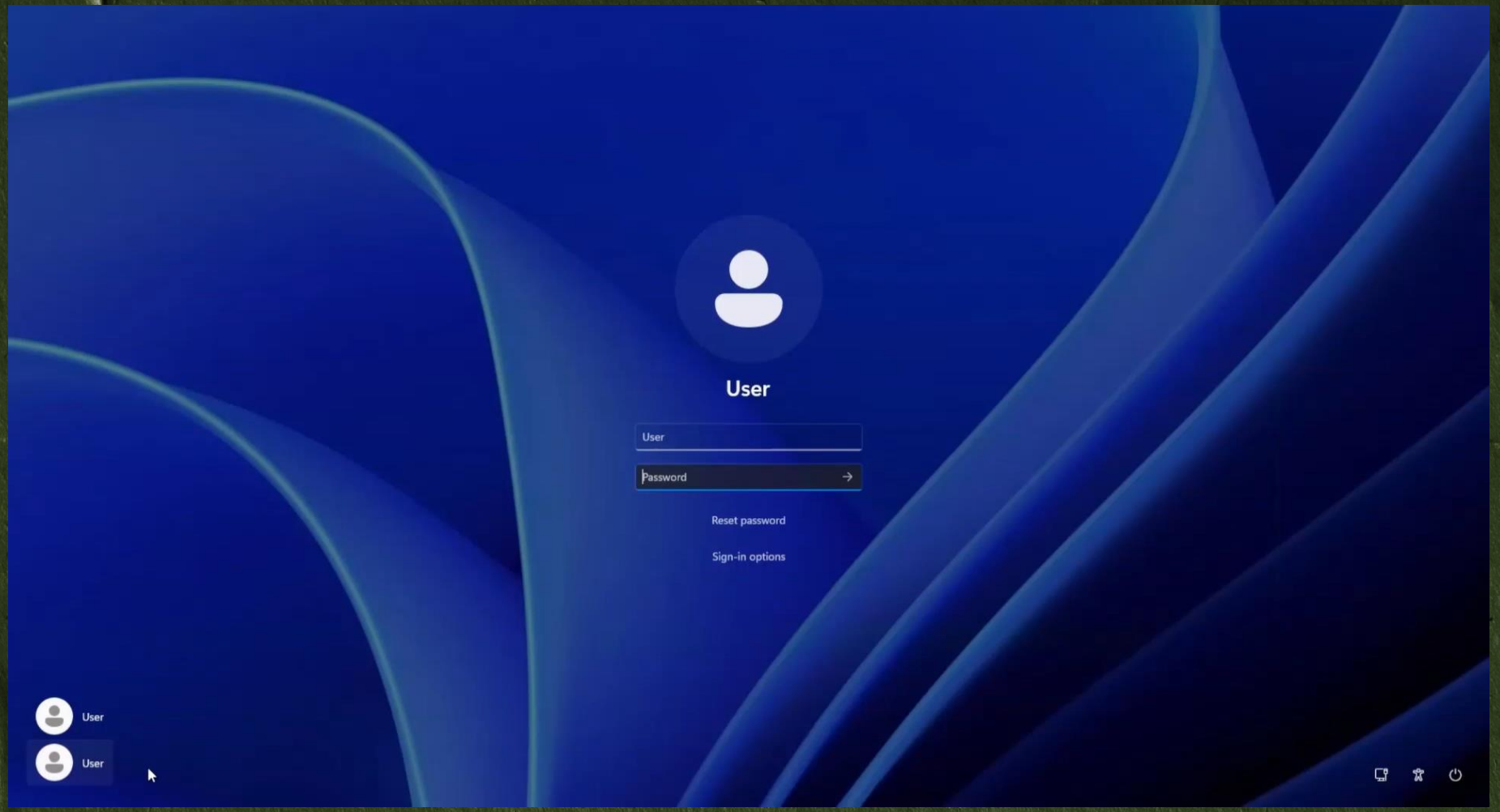
```
if (AssociatedOS == nullptr)
{
    AppendPath(RecoveryDrive,
               L"CBMR",
               CbmrFolderPath,
               szCbmrFolderPath);

    if (PathExists(CbmrFolderPath))
    {
        ShouldLaunchCBMR = TRUE;
    }
}

// [REDACTED SNIP]

if (ShouldLaunchCBMR)
{
    LaunchCBMR();
}
```

Configuration Files Is All We Need

- CBMR loads Reset.ini from the Recovery volume
- This file contains debug hooks that can be used to run arbitrary commands



Demo for Vulnerability #1



BlueHat IL

Our Focus Today

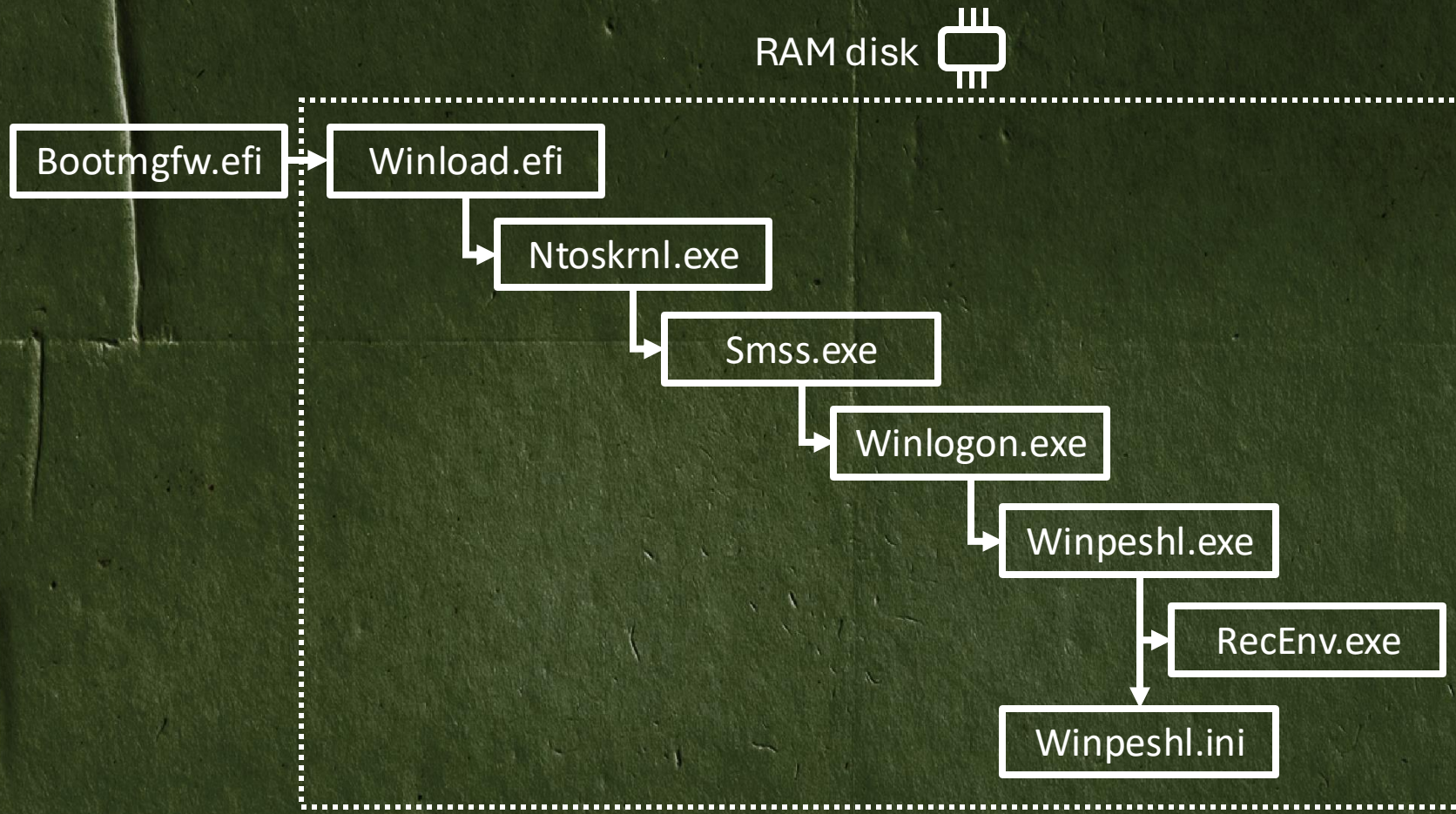
- We will dive into:
  - ✅ CVE-2025-55330 – BitLocker Security Feature Bypass
  -  CVE-2025-55332 – BitLocker Security Feature Bypass

BlueHat IL

WinPE Overview

- WinRE is built on WinPE
- Windows Preinstallation Environment (WinPE) – small Windows OS
- Used for -
  - Deploy captured Windows image
  - Modify existing Windows OS
  - Recover data from unbootable devices
- Custom shell or GUI

WinRE Execution Flow



WinPEShl.ini Launch App

- winpeshl.ini determines which application launches next (typically recenv.exe)

Pseudo Code: App execution

```
void ExecuteStartupApps() {
    bool started = false;

    char* iniPath = LocateConfigFile("winpeshl.ini");

    if (iniPath != NULL) {
        // Case 1: Single app from [LaunchApp] section
        char* app = ReadIniValue(iniPath, "LaunchApp", "AppPath");
        if (app) {
            started = ExecuteApp(app);
        }

        // [REDACTED SNIP]

        // Case 2: INI found but nothing started -> error prompt
        if (!started)
            ExecuteApp("cmd.exe", "/k \"echo <bad INI error>\"");
    }
}
```

- winpeshl.exe attempts to load winpeshl.ini

Pseudo Code: App execution

```
void ExecuteStartupApps() {
    bool started = false;

    char* iniPath = LocateConfigFile("winpeshl.ini");

    if (iniPath != NULL) {
        // Case 1: Single app from [LaunchApp] section
        char* app = ReadIniValue(iniPath, "LaunchApp", "AppPath");
        if (app) {
            started = ExecuteApp(app);
        }

        // [REDACTED SNIP]

        // Case 2: INI found but nothing started -> error prompt
        if (!started)
            ExecuteApp("cmd.exe", "/k \"echo <bad INI error>\"");
    }
}
```

- If valid, the specified application is launched

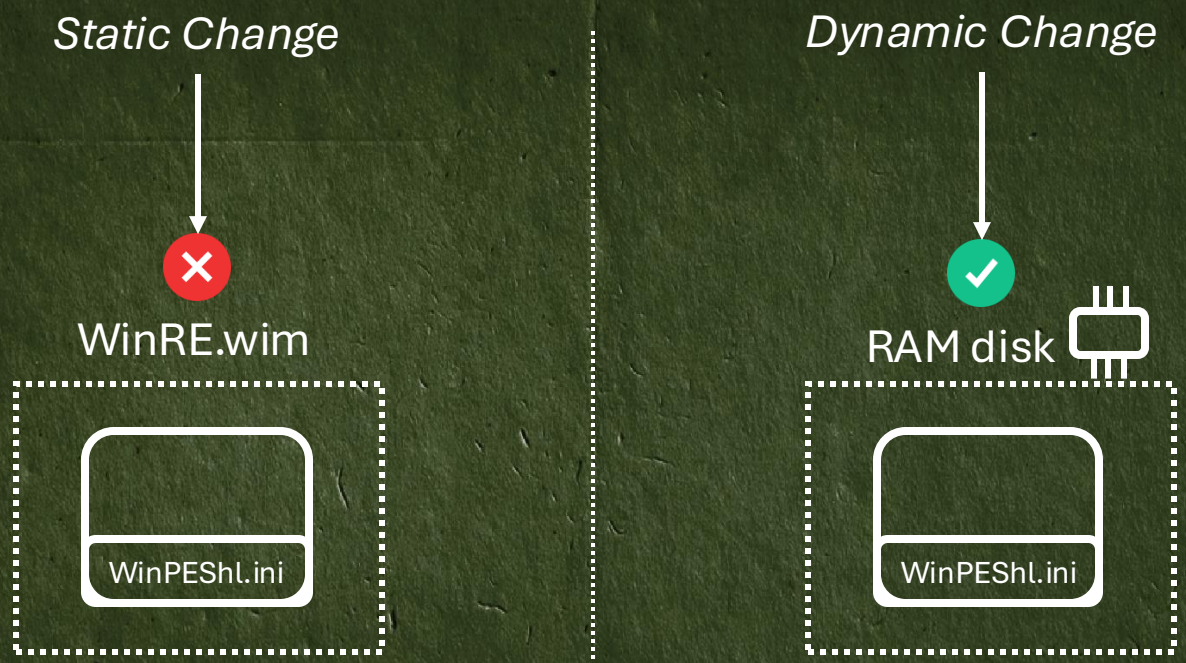
Pseudo Code: App execution

```
void ExecuteStartupApps() {
    bool started = false;

    char* iniPath = LocateConfigFile("winpeshl.ini");

    if (iniPath != NULL) {
        // Case 1: Single app from [LaunchApp] section
        char* app = ReadIniValue(iniPath, "LaunchApp", "AppPath");
        if (app) {
            started = ExecuteApp(app);
        }

        // [REDACTED SNIP]

        // Case 2: INI found but nothing started -> error prompt
        if (!started)
            ExecuteApp("cmd.exe", "/k \"echo <bad INI error>\"");
    }
}
```

- malformed INI triggers cmd.exe launch
- OS volume remains unlocked – can be used to bypass BitLocker

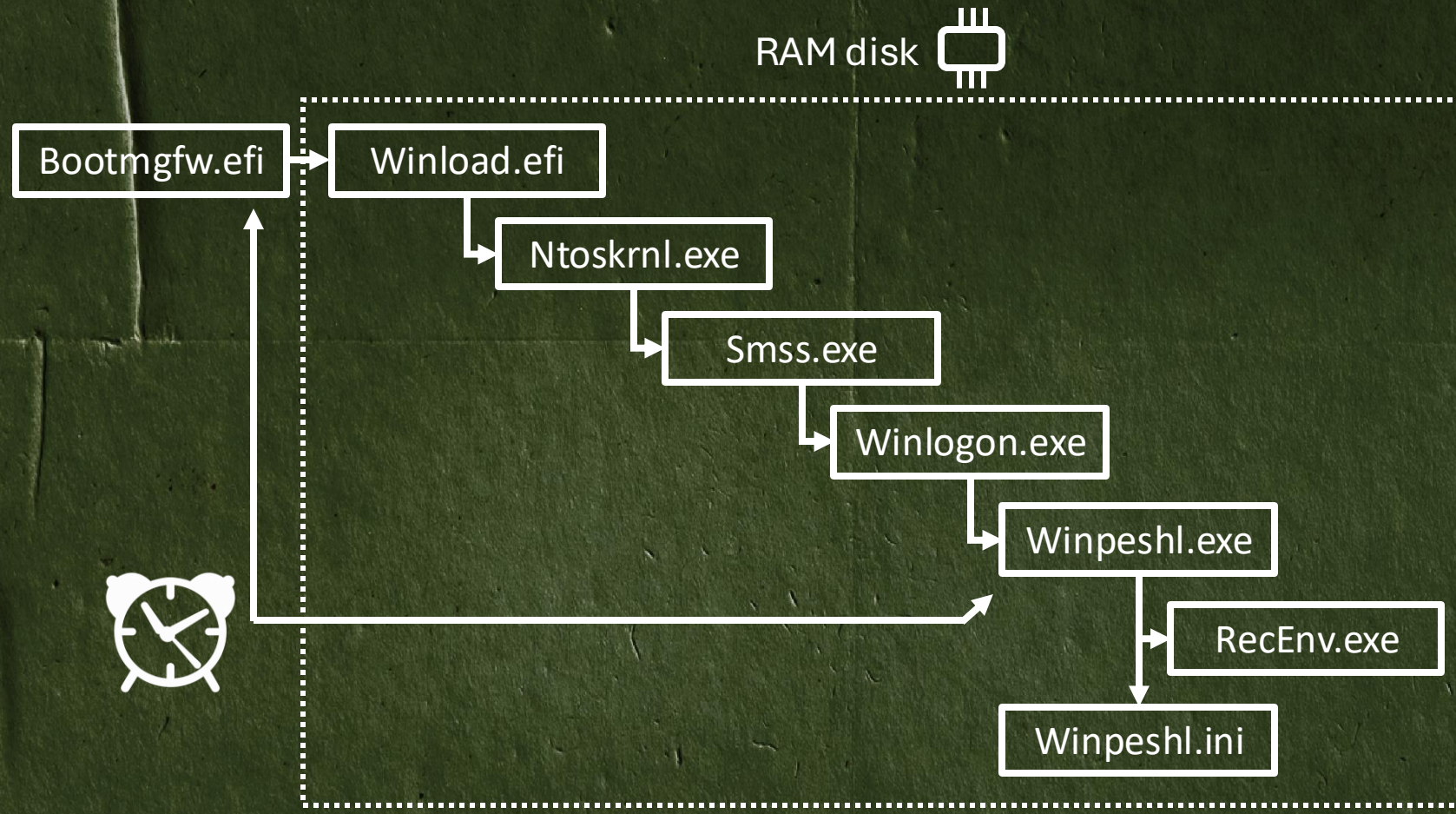
Our Target: Dynamic File Corruption Primitive

- Winpeshl.ini resides in WinRE.wim
  - On-disk modification – impossible due to Trusted WIM boot
  - RAM disk modification – requires a post-boot file corruption primitive



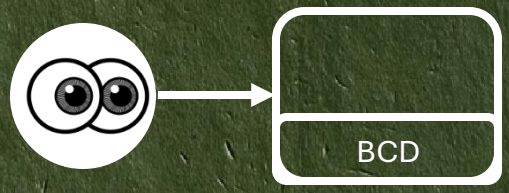
The Opportunity Window

- After bootmgfw.efi, before winpeshl.exe



Where Do We Look?

- Boot Configuration Data (BCD) is a great candidate
- There are more then 250 BCD elements!
- We manually inspected the usage of elements that:
  - Contain a file path
  - Likely not boot-critical
  - Likely written to during boot



Potential Candidate: bootstatfilepath

- The bootstatfilepath element controls the BootStat.dat file path

0x12000044	file path for boot status data log		string	6.0 and higher
	BcdLibraryString_BsdLogPath			6.2 and higher
	BCDE_LIBRARY_TYPE_BSD_LOG_PATH	**bootstatfilepath**		1803 and higher

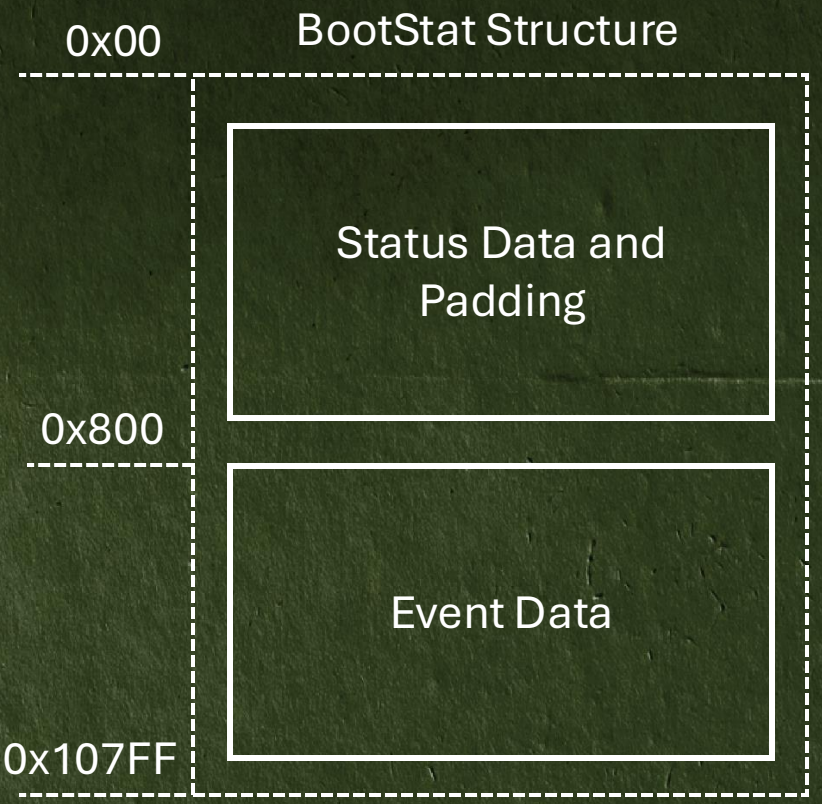
[* Geoff Chappell - BCD Elements]

BootStat.dat Instances

- There are two BootStat.dat instances:
  - Boot Manager Instance: logged to only by Boot Manager
  - OS instance: logged to by Boot Loader, Kernel and System Processes
- The bootstatfilepath BCD element is only applicable for the OS instance

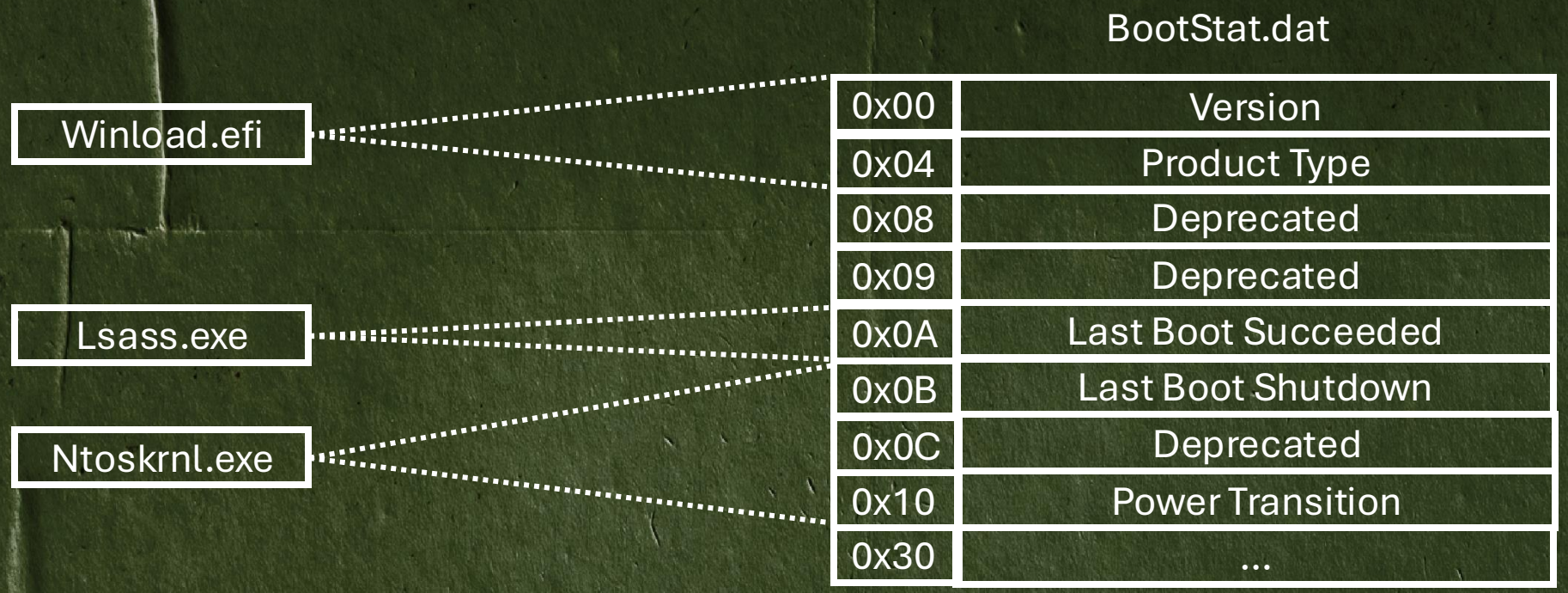
OS Instance BootStat.dat Format

- The OS instance BootStat.dat is separated to two separate sections
  - 0x00 – 0x800 – Status Data and Padding
  - 0x800 – 0x107FF – Event Data



What Do We Care About in BootStat.dat

- WinPEShl.ini size is 0x30 – we only care about status data from 0x00 to 0x30
- The components writing to 0x00 to 0x30 BootStat.dat events are Winload.efi, Ntoskrnl.exe and Lsass.exe



Boot Phase BootStat.dat Writing

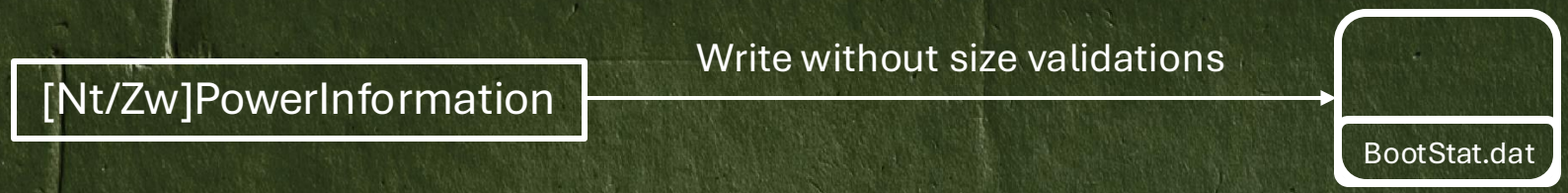
- Winload.efi validates that BootStat.dat is of an expected size
- WinPEShl.ini is much smaller than expected, so all Winload.efi writes will fail

Log Read and Write API (Pseudo Code)

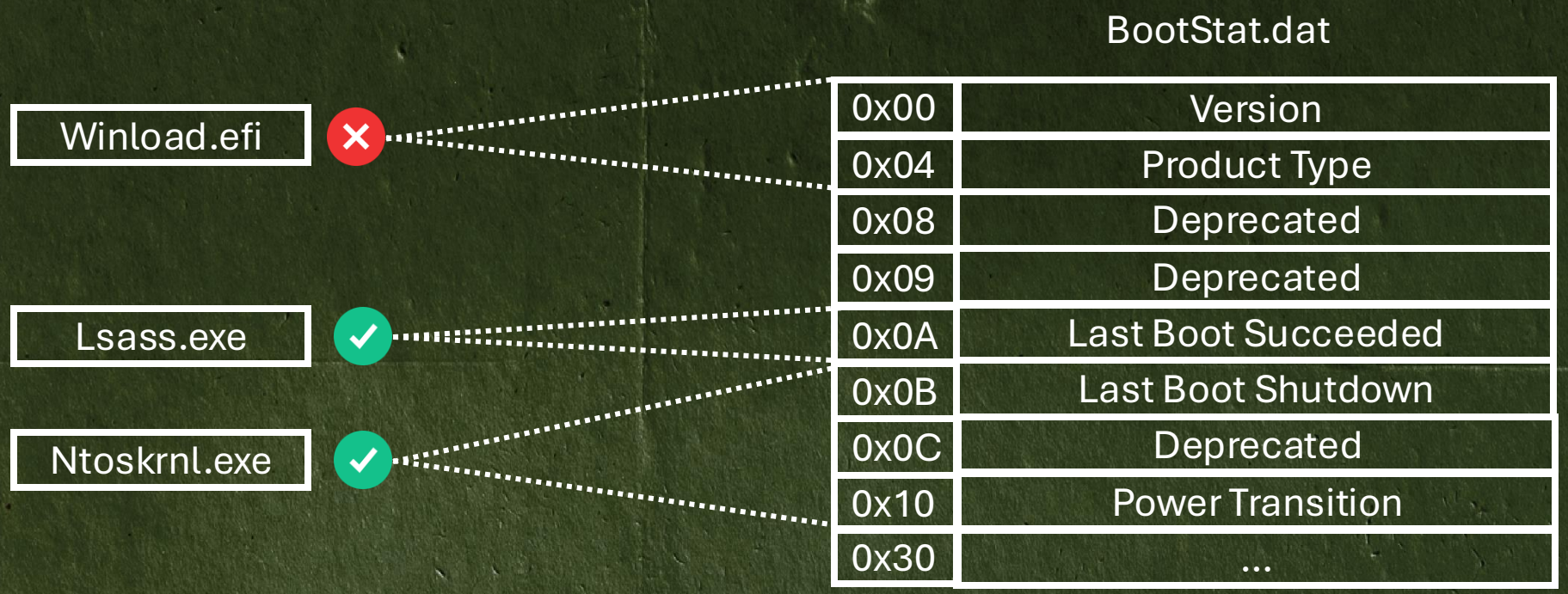
```
// [SNIP REDACTED]
if (BsdFileSize < sizeof(BSD_MIN_FILE_RECORD)) {
    Status = STATUS_BSD_MIN_SIZE_VIOLATION;
    goto BsdMinSizeCheckEnd;
}
// [SNIP REDACTED]
```

OS Phase BootStat.dat Writing

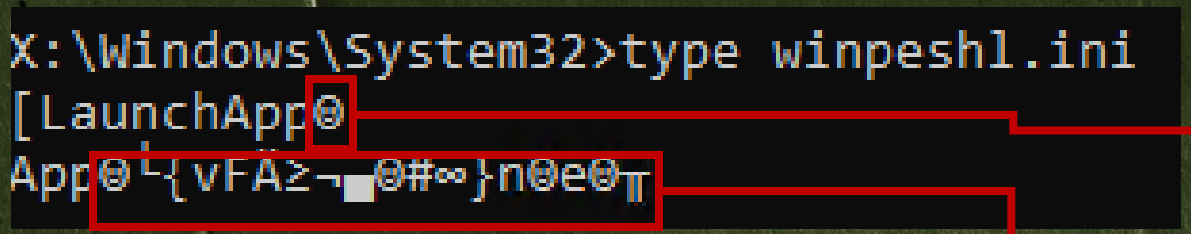
- Kernel and User Mode process writing to BootStat.dat are using the [Nt/Zw]PowerInformation system call
- This system call doesn't perform validations before writing to BootStat.dat



How Does it Look Like?



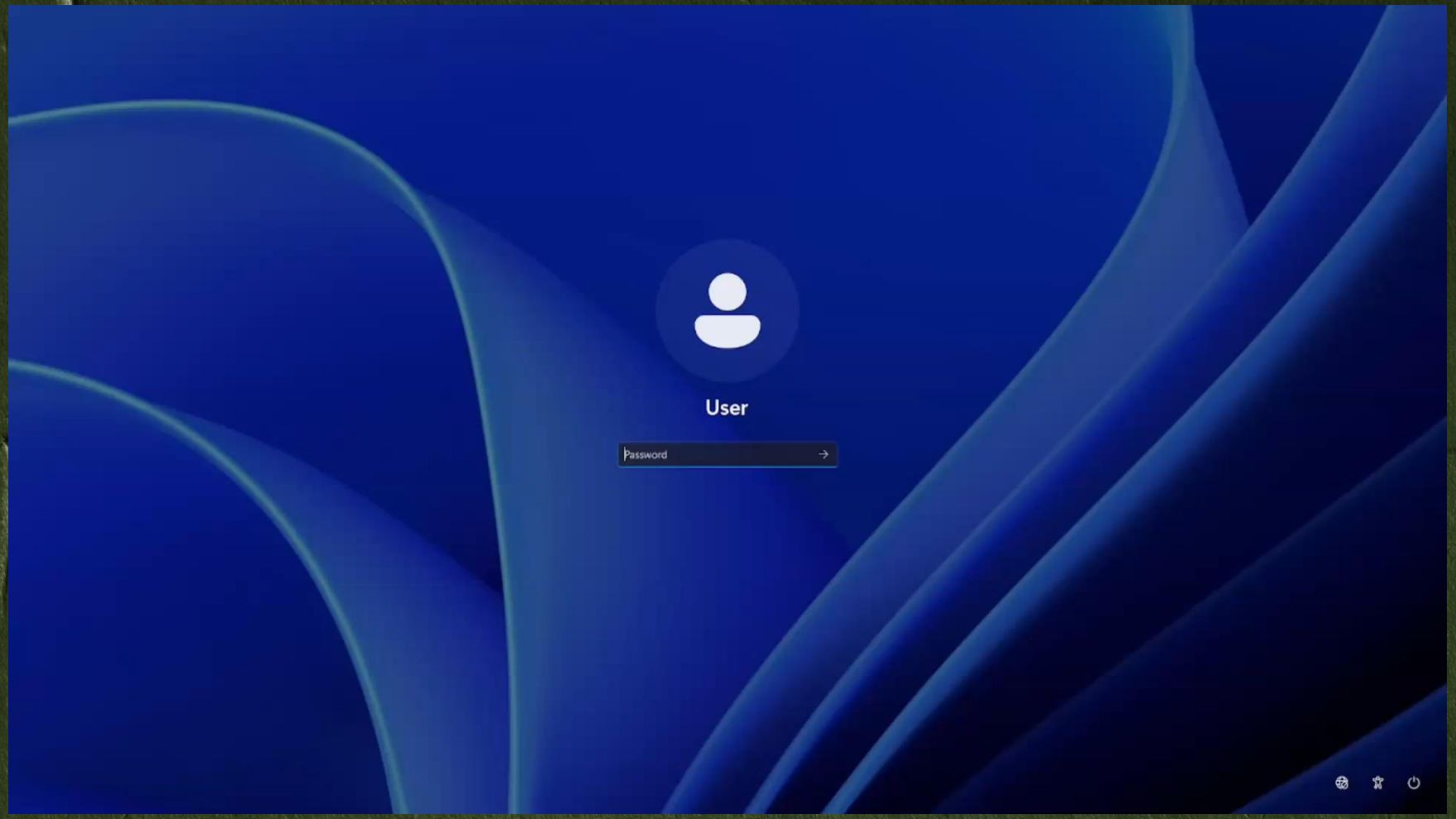
How Does it Look Like?



BootStat.dat

0x00	Version
0x04	Product Type
0x08	Deprecated
0x09	Deprecated
0x0A	Last Boot Succeeded
0x0B	Last Boot Shutdown
0x0C	Deprecated
0x10	Power Transition
0x30	...

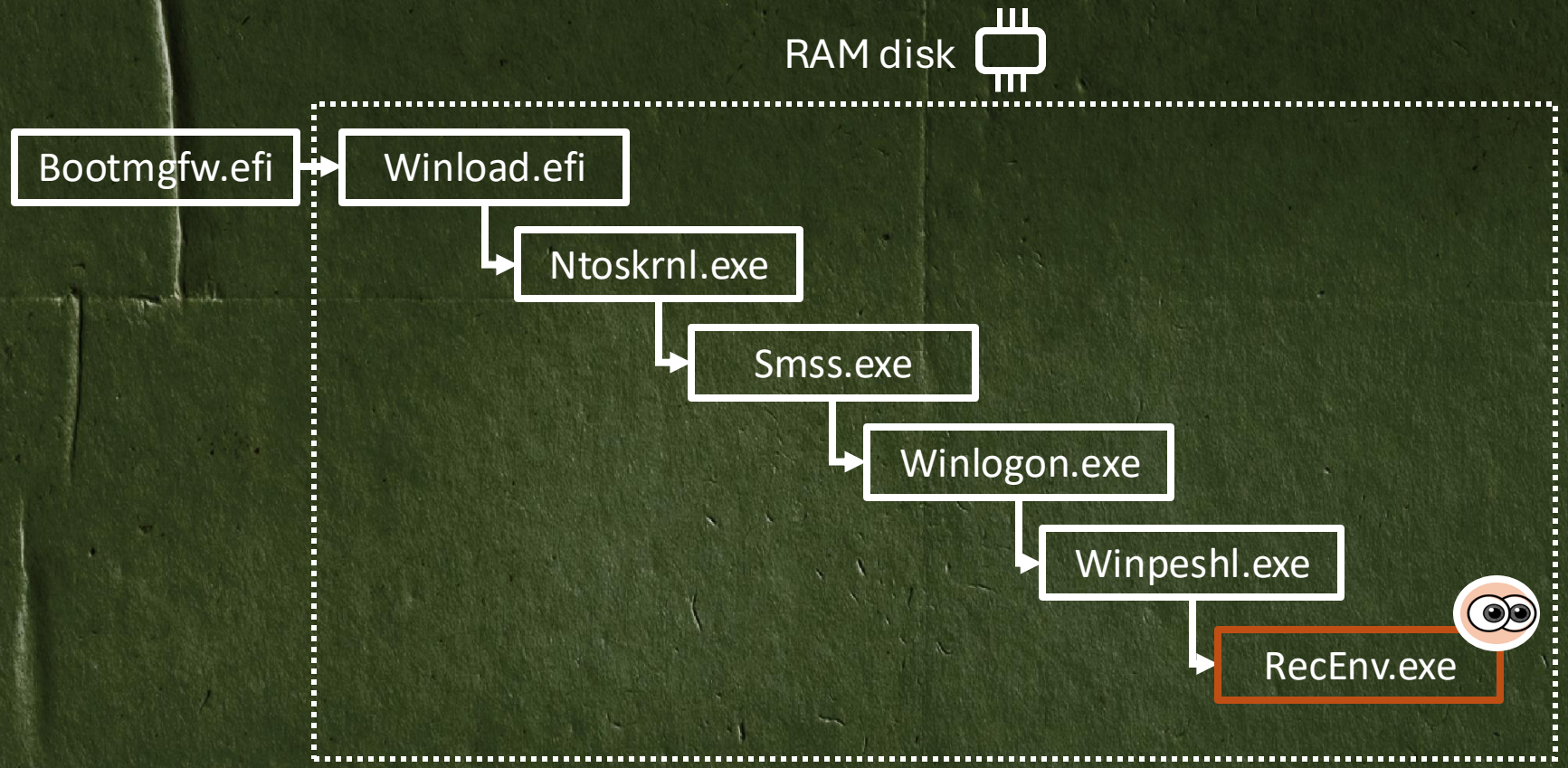
Demo for Vulnerability #2



BlueHat IL

Debugging WinRE

- Our debugging target: RevEnv.exe



RecEnv.exe Debugging Options

- Kernel debugging and attaching to RecEnv.exe
  - Requires turning off security features
  - Requires inserting the BitLocker recovery key every time
- Targeted user-mode debugging via DbgSrv.exe
  - Works without turning off security features

RecEnv.exe Debugging via DbgSrv

- **Run these commands from logged-in OS, as Administrator**

  - mkdir c:\mnt
  - reagentc /mountre /path C:\mnt
  - reg load HKLM\tempmount C:\mnt\windows\system32\config\system
  - reg add HKLM\tempmount\Setup /v CmdLine /t REG_SZ /d "cmd.exe" /f
  - reg unload HKLM\tempmount
  - reagentc /unmountre /path C:\mnt /commit
  - reagentc /disable
  - reagentc /enable
  - reagentc /boottore
  - shutdown –r -t 0

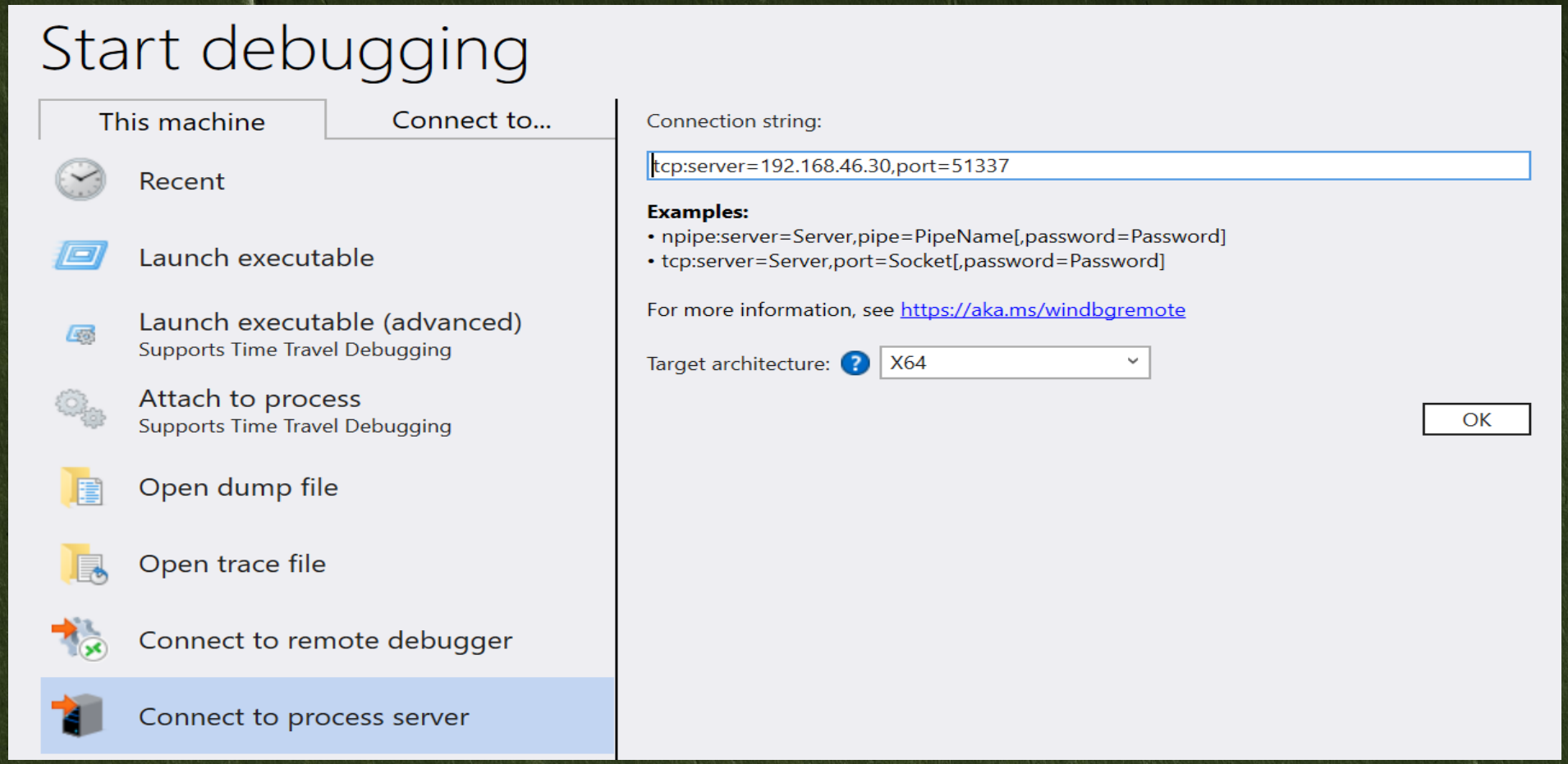
RecEnv.exe Debugging via DbgSrv

- Run these commands from WinRE spawned Command Prompt

  - wpeinit
  - wpeutil disablefirewall
  - D:\DbgSrv.exe –t tcp:port=51337

RecEnv.exe Debugging via DbgSrv

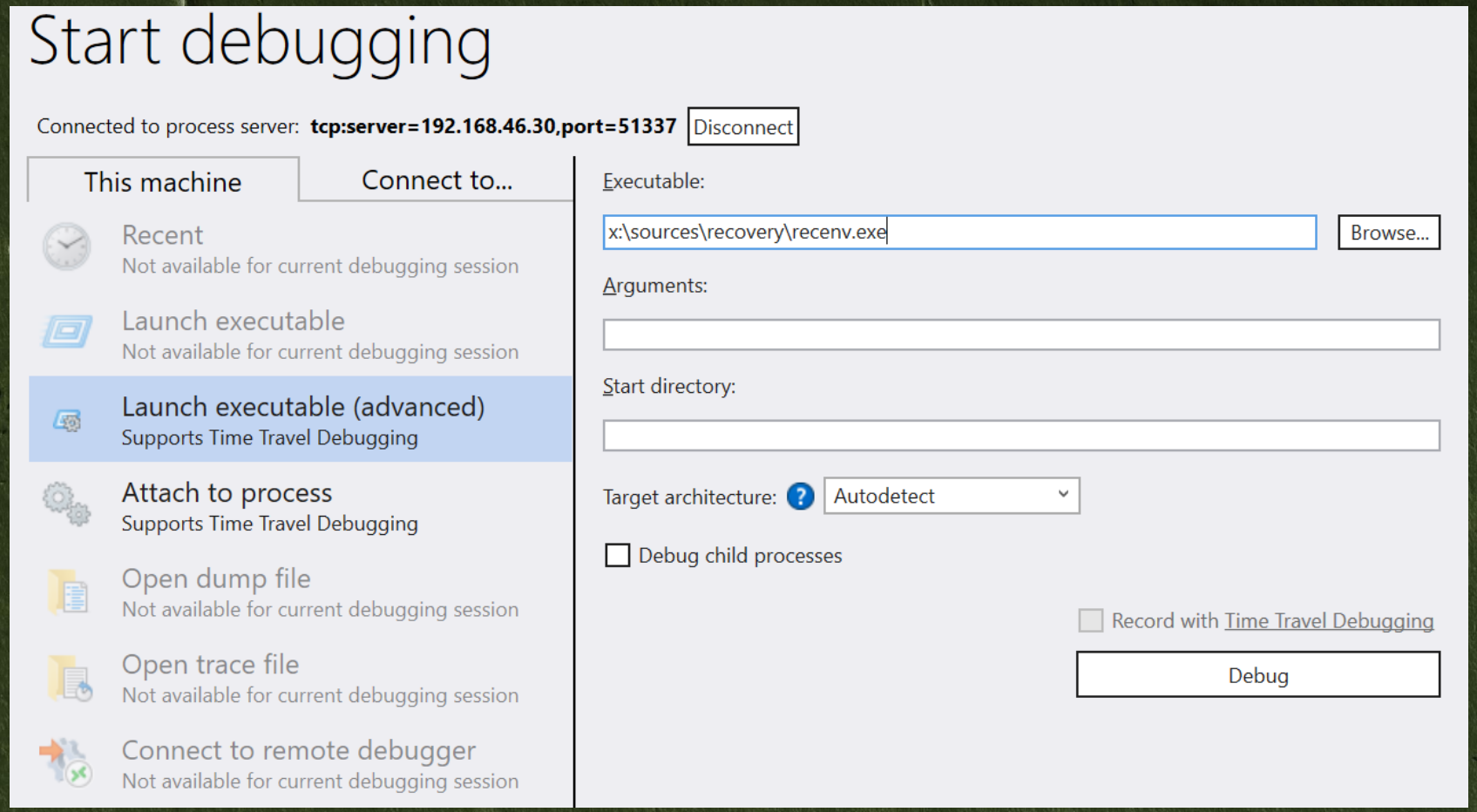
- Connect to the DbgSrv via WinDbg



BlueHat IL

RecEnv.exe Debugging via DbgSrv

- Launch RecEnv.exe via WinDbg



Research Findings and Results

- We found 11 BitLocker bypasses originating from WinRE
  - CVE-2025-48001, CVE-2025-48003, CVE-2025-48800, CVE-2025-48804, CVE-2025-48818, CVE-2025-55330, CVE-2025-55332, CVE-2025-55333, CVE-2025-55337, CVE-2025-55338, CVE-2025-55682
- Fixes were shipped in July and October Patch Tuesday's
- Fuzzers were implemented to fuzz parsing of attacker-facing inputs
- Hardenings were implemented to harden and decrease exposed attack surfaces

Key Takeaway

Overlooked components can hide high-value security research opportunities

Thank You

Security Testing & Offensive Research at Microsoft (STORM)

Alon Leviev (@alon_leviev)
Senior Security Researcher @ Microsoft

Netanel Ben Simon (@NetanelBenSimon)
Senior Security Researcher @ Microsoft

Microsoft

STORM

BlueHat IL