presented by

binarly

Signed and Dangerous: BYOVD Attacks on Secure Boot

UEFI 2025 Developers Conference & Plugfest

October 9, 2025

Presented by:
Alex Matrosov, Fabio Pagani

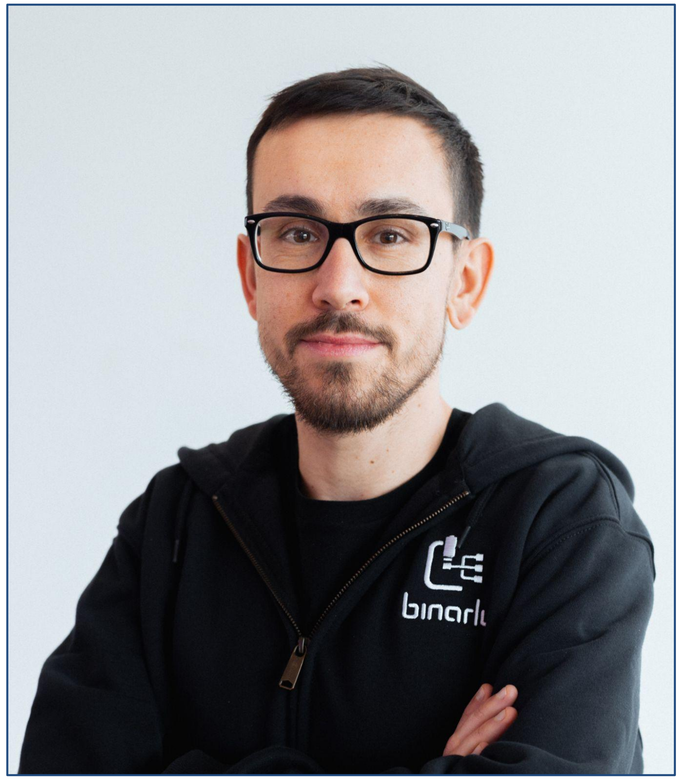
Meet the Presenters

Alex Matrosov

CEO & Head of Research

Alex Matrosov is CEO and Founder of Binarly Inc. where he builds an AI-powered platform to protect devices against emerging firmware threats. Alex has more than two decades of cybersecurity experience. He served as Chief Offensive Security Researcher at Nvidia and Intel Security Center of Excellence (SeCoE). Alex is the Author of numerous research papers and the bestselling award-winning book “Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats”. He is a frequently invited speaker at security conferences, such as REcon, Black Hat, Offensivecon, WOOT, DEF CON, and many others. Additionally, he was awarded multiple times by Hex-Rays for his open source contributions to the research community.

Meet the Presenters



Fabio Pagani

Vulnerability Research Lead

Fabio Pagani is a Vulnerability Research Lead at Binarly, where he works at the intersection of static and dynamic analysis techniques to help secure the UEFI ecosystem. As part of the Binarly REsearch team, he discovered LogoFAIL and helped affected vendors to identify and mitigate this vulnerability. Fabio is always on the lookout for new and impactful firmware vulnerabilities. He also maintains strong connections with the academic community, serving on the program committees of security conferences such as USENIX Security and WOOT.

Agenda

- BYOVD Attacks (UEFI version)
- Taxonomy of Attacks Against Secure Boot
- Finding Secure Boot Bypasses
- Hardening the UEFI Shell
- Mitigations in the UEFI ecosystem
- Conclusions
- Questions

Introduction to BYOVD

- Technique that exploits vulnerabilities in legitimate Windows kernel drivers to gain privileged access
- The drivers are signed and trusted by the OS:
  - Attacker installs the vulnerable kernel driver
  - The vulnerability is exploited in kernel context
  - Profit (?)
- Historically used only by Advanced Persistent Threats (APTs), BYOVD is now found in commodity threats too (ransomware)

https://blog.talosintelligence.com/exploring-vulnerable-windows-drivers/

BYOVD + UEFI = ?

- UEFI firmware also relies on signature verification when Secure Boot is active
- Secure Boot: only trusted and verified modules are allowed to be executed
- Determination based on the content of NVRAM variables:
  - db → allowed signatures
  - dbx → revoked signatures

What is the impact of BYOVD on UEFI?

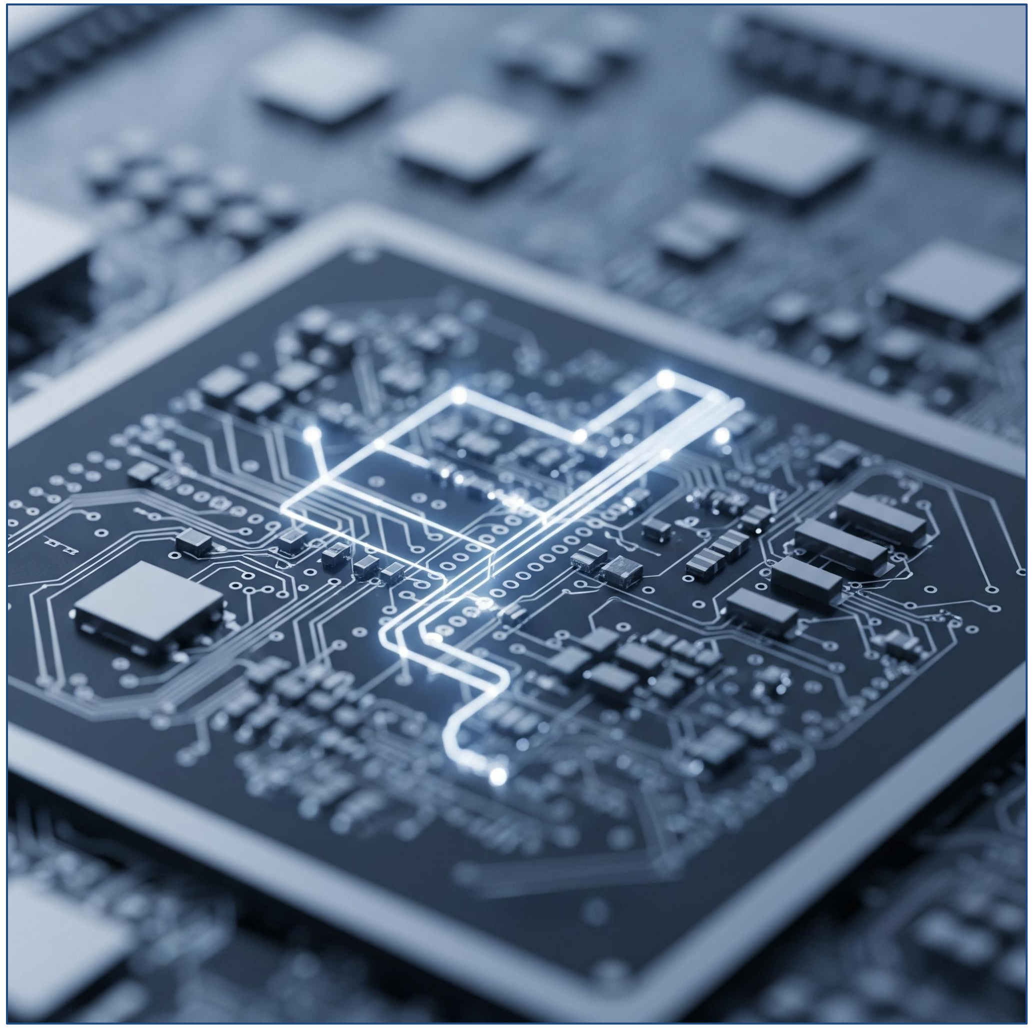
Taxonomy of Attacks Against Secure Boot

1. Double-use modules: Trusted programs exposing a functionality that can be misused to run untrusted code (e.g. the UEFI Shell)
2. Trusted but vulnerable modules: Trusted programs that contain exploitable vulnerabilities (e.g. CVE-2025-3052)
3. Leaked private keys: Keys used in authentication that are compromised, allowing attackers to sign malicious modules (e.g. PKfail)
4. Verification logic bugs: Bugs in the verification process itself that allows an attacker to bypass verification (e.g. CVE-2025-6198)
5. Debug or incomplete features: Features intended for debugging end up in production devices and allow to bypass authentication (e.g. CVE-2021-0114)

Identify BYOVD in the UEFI ecosystem

High-level plan to identify double-use and trusted but vulnerable modules:

1. Collect a comprehensive dataset of UEFI modules
2. Determine which modules are trusted by real-world firmware
3. Scan trusted modules to detect double-use and trusted but vulnerable modules



Large database of UEFI modules

- Sources:
  1. Internal collection of UEFI firmware (gathered over 5+ years)
  2. Private telemetry data (pk.fail detector)
  3. Public threat intelligence feeds (VirusTotal)
- Indexed over 10 million modules

	name	guid	hex(hash)	authenticode	length(cert)
1	RealtekUndiDriver	e88db748-a947-46cf-ab6f-5c99b6c6c4b8	E70AD86ED34F1E7948253B4AB7F18…	FC5C7711F42C178A03C2B5067DED60C96BD9…	8672
2	RealtekPxe	1be14579-d805-4c3b-8874-410b818674e9	A4782AD88B9AA789F2C6421FB0C90…	BFD73544D17BEAB0ABB26C28335D3141C403…	8640
3	InfineonTpmUpdateDxe	8900e28f-de99-4fc4-894b-6f41cd139a48	CE383755FB2B13984C6750791495A…	E39214F6C5F4E1C7653640B3D25DE9036837…	8632
4	A8DAFB9B-3529-4E87-8584-ECDB6A5B78B6	a8dafb9b-3529-4e87-8584-ecdb6a5b78b6	46244EE2B5FDC63A0DD05C021A6EA…	B9CE1967709E788BC85D709F9A324D7C54E5…	8552
5	RtkUsbUndiDxe	3ed432c9-5f9d-415d-a1c3-2b0427a90758	ACB9A6CDDC57B623AD939891C9C06…	E822EE1DB8F068696FD106295EADCA7F5393…	8552
6	7C0B621C-118C-49F3-BA6A-003244829342	7c0b621c-118c-49f3-ba6a-003244829342	5CDF3D75C0EC0800B9692AEDEF195…	3789CA5B6CCD21A528374F0FB85958516966…	1424
7	RtkUndiDxe	b7b82ad8-3349-4968-a940-7b8c265ff9b4	1E8ABB2E42F4F9D041CCC71DB642A…	1ABC75968C86E2DA5F9EAE4187A689D3EE47…	8744
8	AEB1671D-019C-4B3B-BA00-35A2E6280436	aeb1671d-019c-4b3b-ba00-35a2e6280436	36D5DD7D857FF7A9CBCE64EEEAFB6…	B09EAAADCE7C95318364D4A0103EAB08DEFC…	20760
9	Rtk8111UndiBin	2851e234-20fd-4d1e-9041-dcb8f3025cae	6E2DD29F159EDF01187FB6B518DBA…	F27308D9AB25BEADD7413A19E7E5232B5DF2…	9624
10	EzFlashInterfaceBin	d1531968-e138-4e2e-8f7e-383307169276	C33B9914C7D8FB5767B733FE121C5…	0FAC038F39EC874CF1D5CB56E188806B21A2…	1408

Which UEFI Modules Are Trusted?

- Selected recent firmware images, covering most OEMs
- Identified which modules from the database are trusted by the selected firmware images
- Results:
  - Discovered 7,157 unique modules trusted by recent firmware
  - On average firmware trusts 1,500 modules with peaks over 4,000 modules

A vulnerability in any trusted module can be used to bypass Secure Boot on the device

Trusted but Vulnerable Modules

- Scanned modules with our platform to uncover issues in NVRAM variable handling and beyond
- Automatically identified one vulnerability (CVE-2025-3052) in a module signed with the Microsoft's third-party UEFI certificate
- June Patch Tuesday: Microsoft added 14 modules to dbx

```
RT->GetVariable(L"IhisiParamBuffer", GUID, 0LL, &Size, &VarContent)
...
VarContent->param3 = 0LL;
VarContent->param5 = 0LL;
VarContent->param6 = 0LL;
VarContent->param1 = 0x83EFLL;
VarContent->param2 = '$H2O';
VarContent->param4 = 0xB2LL;
...
```

VarContent is blindly trusted and used for multiple memory writes!

Double-Use Modules

- Focus on UEFI Shell: isolated incidents or ecosystem-wide issue?
- Large attack surface, dangerous commands (*mm*) and scripts executed at startup (*startup.nsh*)

```
Shell> dmem 0x11223344 20
Memory Address 0000000011223344 20 Bytes
  11223344: 00 00 00 00 00 00 00 00-00 00 00 00  *................*
  11223354: 00 00 00 00 00 00 00 00-00 00 00 00  *................*

Shell> mm 0x11223344 DDCCBBAA -w 4

Shell> dmem 0x11223344 20
Memory Address 0000000011223344 20 Bytes
  11223344: AA BB CC DD 00 00 00 00-00 00 00 00  *................*
  11223354: 00 00 00 00 00 00 00 00-00 00 00 00  *................*
```

Double-Use Modules

- Focus on UEFI Shell: isolated incidents or ecosystem-wide issue?
- Large attack surface, dangerous commands (*mm*) and scripts executed at startup (*startup.nsh*)
- Discovered 30 UEFI shells trusted by hundreds of devices
  - 29 shells are signed with an OEM certificate present in db
  - 1 shell is trusted because it's Authenticode hash was added to db
- Disclosure with CERT/CC is ongoing!

From Trusted Shell to Untrusted Code Execution

Core idea: use the mm command to overwrite gSecurity2

```
if (gSecurity2 != NULL) {
  //
  // Verify File Authentication through the Security2 Architectural Protocol
  //
  SecurityStatus = gSecurity2->FileAuthentication (
                              gSecurity2,
                              OriginalFilePath,
                              FHand.Source,
                              FHand.SourceSize,
                              BootPolicy
                              );
  if (!EFI_ERROR (SecurityStatus) && ImageIsFromFv) {
    //
```

When gSecurity2 is NULL, Secure Boot is not enforced!

From Trusted Shell to Untrusted Code Execution

We developed and tested a PoC:

1. From a privileged OS shell:
   - Copy the trusted UEFI shell and a *startup.nsh* script to the EFI System Partition
   - Place a second unsigned UEFI module (the payload) on the partition
   - Configure the Boot Manager to run the UEFI shell before the unsigned module

From Trusted Shell to Untrusted Code Execution

We developed and tested a PoC:

2. After rebooting the device:
   - The Boot Manager runs the UEFI shell
   - The UEFI shell automatically executes *startup.nsh*, which issues an *mm* command to zero *gSecurity2*
   - The unsigned module containing the malicious payload executes successfully


binarly
Combining a Secure Boot Bypass with a Bootkit on Windows 11

Hardening the UEFI Shell

Commit f881b4d

kraxel authored and **mergify[bot]** committed on Feb 25, 2024

```
OvmfPkg: only add shell to FV in case secure boot is disabled

The EFI Shell allows to bypass secure boot, do not allow
to include the shell in the firmware images of secure boot
enabled builds.

This prevents misconfigured downstream builds.

Ref: https://bugs.launchpad.net/ubuntu/+source/edk
Ref: https://bugzilla.tianocore.org/show_bug.cgi?i
Signed-off-by: Gerd Hoffmann <kraxel@redhat.com>
Reviewed-by: Laszlo Ersek <lersek@redhat.com>
Acked-by: Jiewen Yao <Jiewen.yao@intel.com>
Message-Id: <20240222101358.67818-13-kraxel@redhat
```

master (#5406) · edk2-stable202508 ··· edk

https://github.com/tianocore/edk2/commit/f881b4d129602a49e3403043fc27550a74453234

OvmfPkg/Include/Fdf/ShellDxe.fdf.inc +1 -1

```
@@ -2,7 +2,7 @@
2  2    #    SPDX-License-Identifier: BSD-2-Clause-Patent
3  3    ##
4  4
5     - !if $(BUILD_SHELL) == TRUE
   5  + !if $(BUILD_SHELL) == TRUE && $(SECURE_BOOT_ENABLE) == FALSE
6  6
7  7    !if $(TOOL_CHAIN_TAG) != "XCODE5"
8  8    !if $(NETWORK_ENABLE) == TRUE
```

Hardening the UEFI Shell

mediouni-m commented 2 weeks ago

When UEFI Secure Boot is on, we should not allow for the UEFI shell to be used. As such, disabling it in that scenario.

- [ ] Breaking change?
  - **Breaking change** - Does this PR cause a break in build c
  - Examples: Does it add a new library class or move a moc
- [x] Impacts security?
  - The UEFI shell is known as insecure.
- [ ] Includes tests?
  - **Tests** - Does this PR include any explicit test code?
  - Examples: Unit tests or integration tests.

How This Was Tested

Has been on AWS since a while, see
https://github.com/aws/uefi/blob/5c3ac896feea3923a96944dc23e
stable202211/0032-edk2-stable202211-uefi-shell-Disable-the-she
on.patch

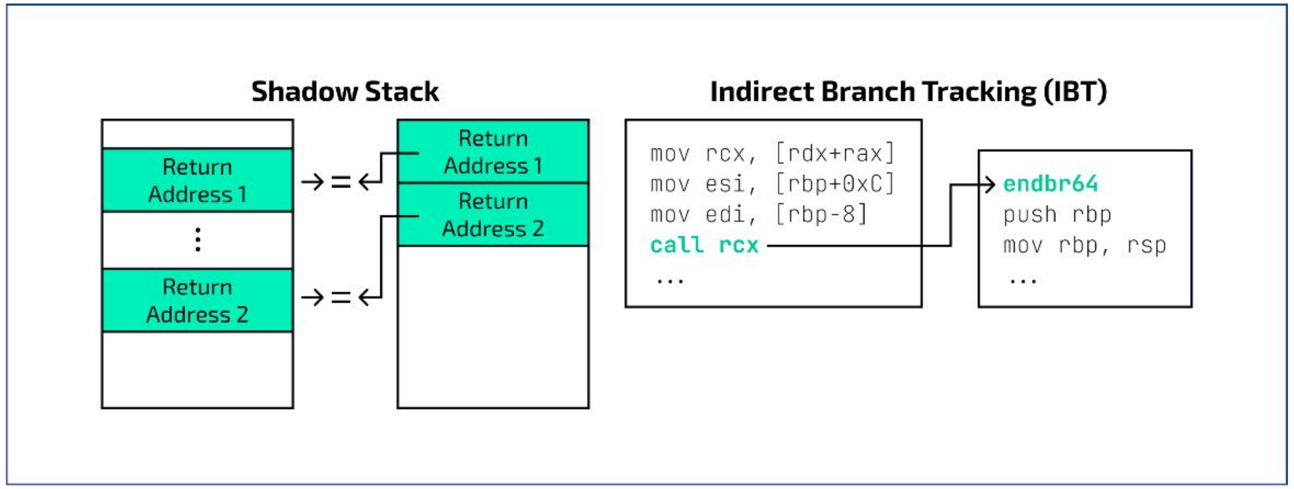
https://github.com/tianocore/edk2/pull/11486

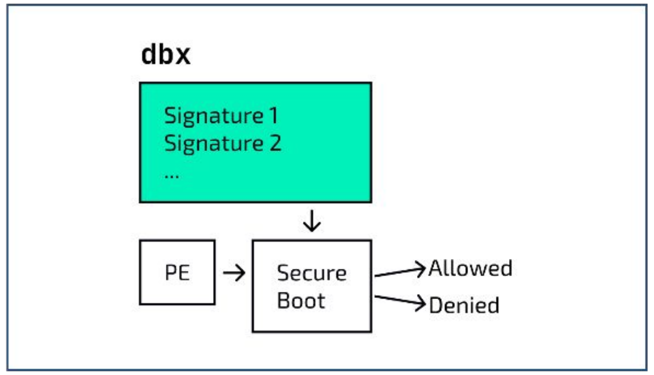
11 ShellPkg/Application/Shell/Shell.c Viewed

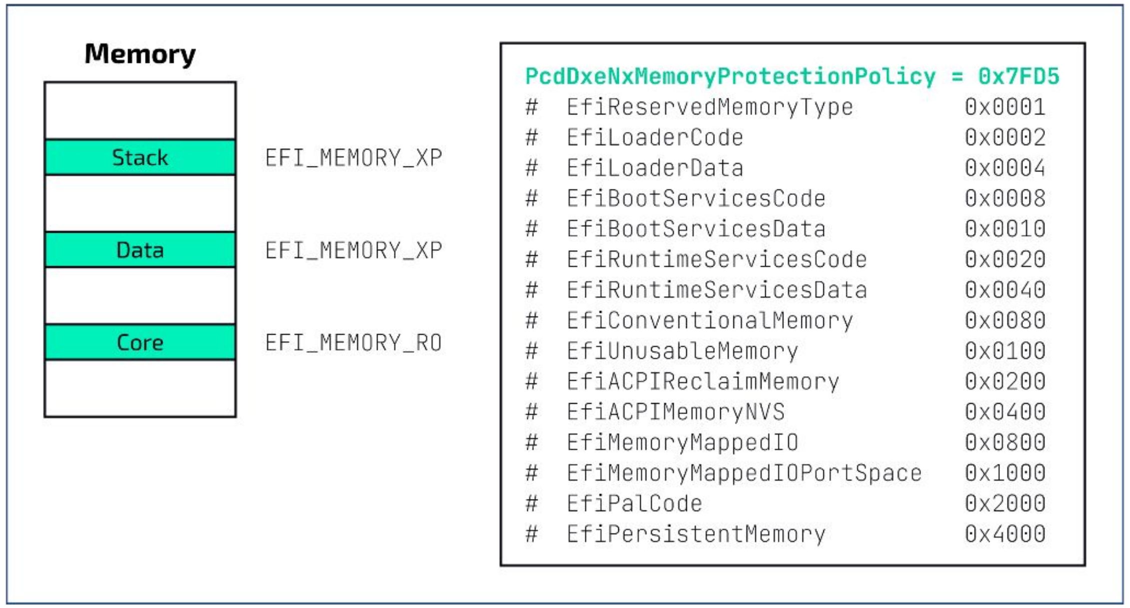
```
@@ -358,6 +358,17 @@ UefiMain (
    EFI_HANDLE                       ConInHandle;
    EFI_SIMPLE_TEXT_INPUT_PROTOCOL   *OldConIn;
    SPLIT_LIST                       *Split;
+   UINT8                            *SecureBoot;
+
+   // If Secure Boot is enabled, do not launch the UEFI shell
+   SecureBoot = NULL;
+   GetEfiGlobalVariable2 (EFI_SECURE_BOOT_MODE_NAME, (VOID **)&SecureBoot,
  NULL);
+   if ((SecureBoot != NULL) && (*SecureBoot == SECURE_BOOT_MODE_ENABLE)) {
+     FreePool (SecureBoot);
+     return EFI_SECURITY_VIOLATION;
+   } else if (SecureBoot != NULL) {
+     FreePool (SecureBoot);
+   }

    if (PcdGet8 (PcdShellSupportLevel) > 3) {
      return (EFI_UNSUPPORTED);
```

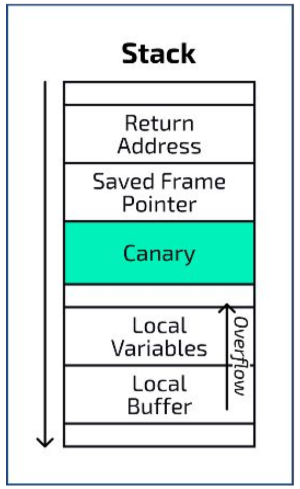
Mitigations REsearch

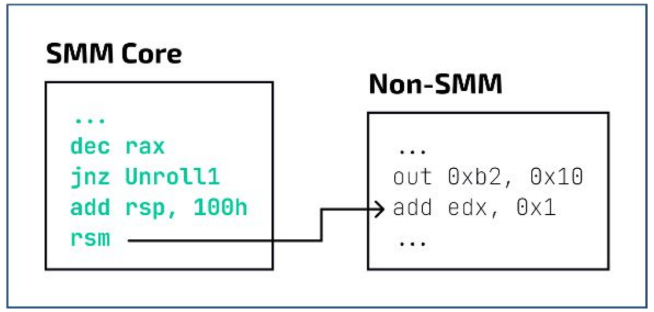
- Why these vulnerabilities can be easily exploited?
- UEFI firmware lacks basic mitigations :(





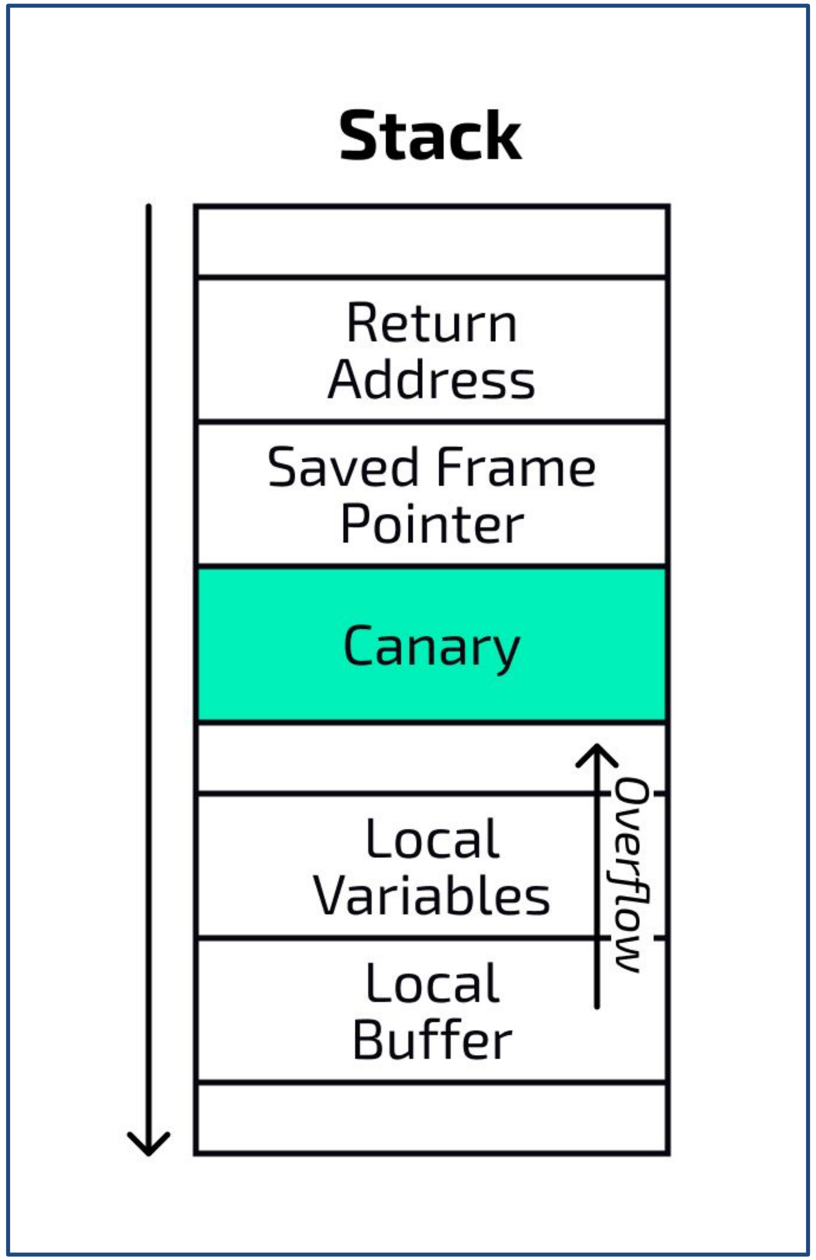






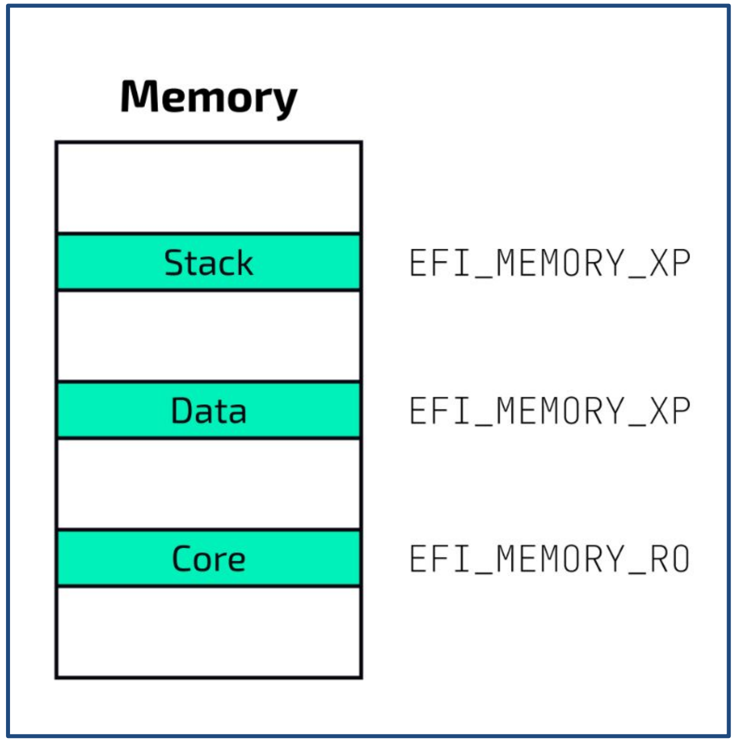
Stack Canaries

- Dynamic stack canaries landed only recently in EDK2 **(Feb 2025)**
- Before this contribution:
  - Stack canaries were enabled **only** for ARM targets compiled with GCC
  - For the remaining targets, canaries were **explicitly** disabled (like X86)



No eXecute (NX)

- Multiple PCDs allow for customization of this mitigation:
  - PcdImageProtectionPolicy: which modules have read-only code and non-executable data sections
  - PcdDxeNxMemoryProtectionPolicy: defines non-executable memory types (e.g. *EfiLoaderData*)
  - PcdSetNxForStack: stack marked as non-executable
- In x86, image protection policy applied **only** to images loaded from firmware volumes, **disabled** for other sources
- In ARM, image protection policy applies to **all** sources, and *NxMemoryProtectionPolicy* protects **all** non-code regions, including the stack



Adoption of Canaries and NX

Stack Canaries:

- Out of 2.3M analyzed modules, only 2,674 (0.12%) use stack canaries
- No x86 firmware includes this basic mitigation

No-Execute (NX):

- Only 10% of analyzed firmware enforce a correct memory protection policy
- In most cases, image protection is misconfigured so bootloaders remain unprotected

Hidden Catch: Section Alignment

- NX enforcement requires the PE section alignment to match the EFI page size (0x1000)

```
// Check RequiredAlignment
if ((RequiredAlignment != NULL) && ((SectionAlignment & (*RequiredAlignment - 1)) != 0)) {
  DEBUG ((
    DEBUG_WARN,
    "!!!!!!!!  Image Section Alignment(0x%x) does not match Required Alignment (0x%x)  !!!!!!!!\n",
    SectionAlignment,
    *RequiredAlignment
    ));

  return EFI_ABORTED;
}
```

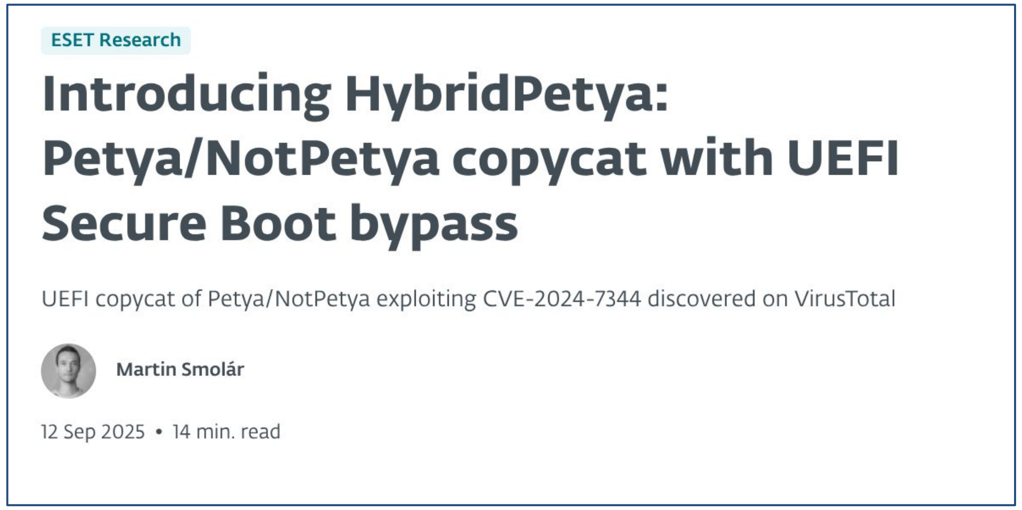
- Critical gap in practice: 68% of DXE modules **fail this requirement**, leaving writable code sections and executable data sections

Conclusions

- Verification of firmware components is complex
- Secure Boot represents a last line of defense against firmware-level threats
- Large number of signed modules in the wild → custom Secure Boot certificates
- Mitigations remain largely absent in the ecosystem, broader adoption is needed
- Are UEFI-level threats coming?

https://www.welivesecurity.com/en/eset-research/introducing-hybridpetya-petya-notpetya-copycat-uefi-secure-boot-bypass/

https://x.com/hasherezade/status/1965389009175412769



Questions?

References

1. https://blog.talosintelligence.com/exploring-vulnerable-windows-drivers/
2. https://www.binarly.io/blog/another-crack-in-the-chain-of-trust
3. https://www.binarly.io/blog/pkfail-untrusted-platform-keys-undermine-secure-boot-on-uefi-ecosystem
4. https://www.welivesecurity.com/en/eset-research/introducing-hybridpetya-petya-notpetya-copycat-uefi-secure-boot-bypass/