The Untold Story of the BlackLotus UEFI Bootkit

Archived: 2026-04-05 14:54:30 UTC

by Alex Matrosov

My experience with the analysis and detection of rootkits and bootkits goes back more than 20 years. In the early 2000s, the main challenge was dealing with infected machines when rootkits and bootkits modified the operating system kernel to conceal malicious components. It was such a fun time reverse engineering advanced threats in the good old days that I co-wrote "Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats," a book full of the most interesting stories of our time going down the rabbit hole of advanced malware.

When the major operating systems moved to UEFI support and Secure Boot was widely implemented, I really believed the golden age of bootkits had ended and the natural evolution would lead in the direction of firmware implants. I was wrong. Last week, researchers at ESET published "BlackLotus UEFI Bootkit: Myth Confirmed," confirming the existence of UEFI bootkits all over again. UEFI bootkits like BlackLotus are not new and have been publicly seen since 2013 or even earlier. Modified or replaced bootloaders create very visible and noisy Indicators of Compromise (IOCs) that attackers who are focused on persistence want to avoid.

Nevertheless, BlackLotus has been detected in-the-wild by the end of 2022, which is surprising for a malware class that has been around for 10+ years.

Dive into (in)secure boot and the CVSS scoring problem

This story also has a supply chain twist. A critical aspect of the BlackLotus story lies in supply chain problems relating to modern operating systems, their bootloaders, and UEFI firmware. In order to bypass Secure Boot at scale, BlackLotus exploits CVE-2022-21894, a vulnerability patched by Microsoft in January 2022. A proof-of-concept exploit was released in August 2022, seven months after Microsoft's public disclosure.

The CVE-2022-21894 vulnerability's CVSS score (4.4 medium) doesn't seem to indicate a serious vulnerability, does it?

The problem with most of the Secure Boot bypass vulnerabilities is that they require local or physical access to the target. This significantly affects CVSS scoring in terms of the impact. Since Secure Boot bypass attacks cross security boundaries or allow the disabling of security features, we need to treat them as privilege escalation vulnerabilities to more accurately reflect the CVSS impact score.

As the operating system bootloader becomes the middle layer between the firmware and operating system during the boot process, the bootloader provides a significant attack surface that can impact a lot of security features like disk encryption and secure boot. Currently, there is little documentation on attacks against bootloaders and a lot of inconsistency in the published existing knowledge. Here are some Microsoft advisories related to Secure Boot bypasses and other bootloader vulnerabilities.

As an aside, the navigation on the Microsoft Security Response Center (MSRC) website is a complete disaster from the perspective of tracking retrospective vulnerabilities and attacks. According to CVSS scores, all of these issues fall under ‘medium-severity’ impact. Despite this, many of them allow bypassing Secure Boot and attacking the bootloader with serious consequences.

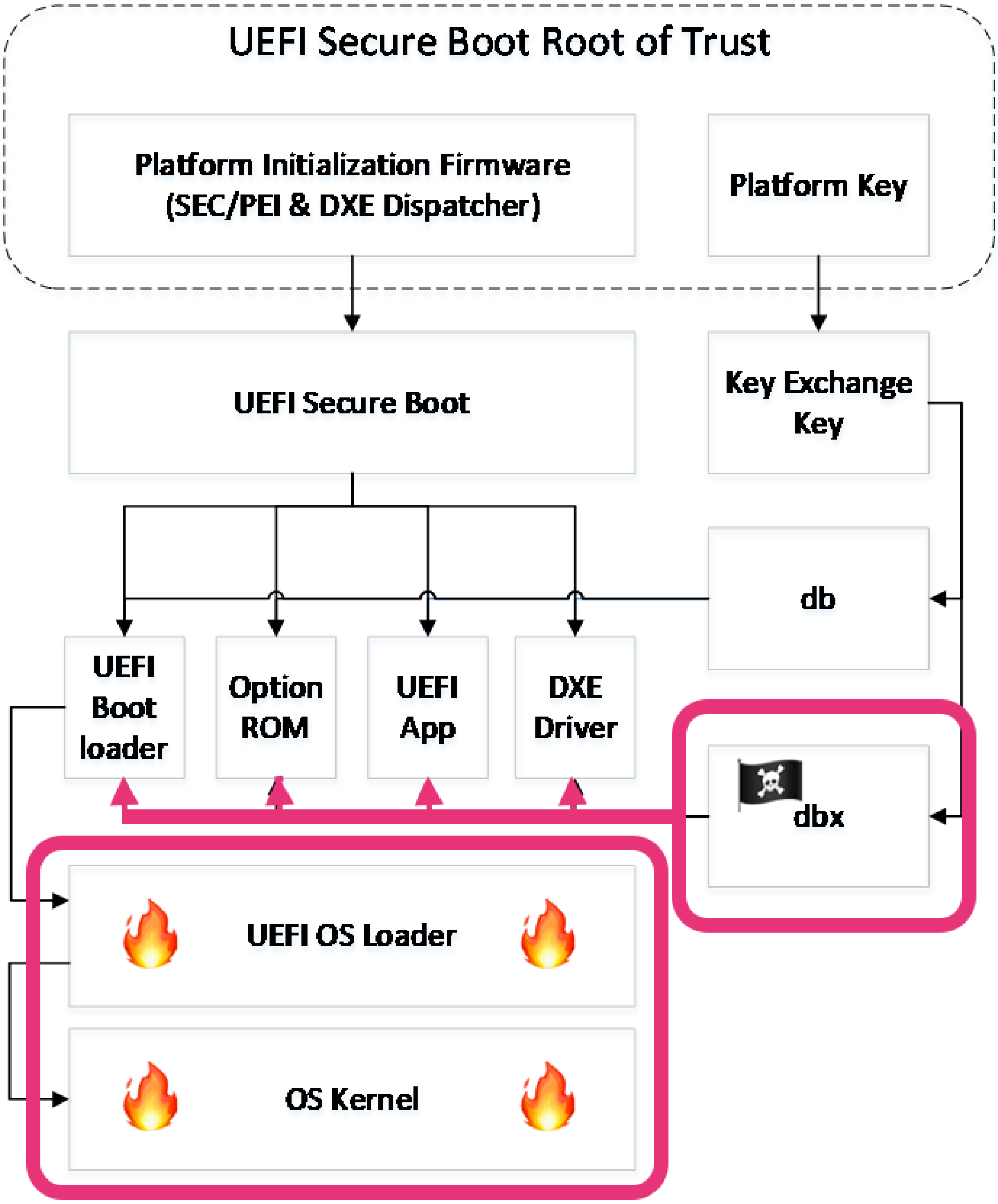
Even after the vendor fixes the secure boot bypass vulnerabilities shown in the figure above, the vulnerabilities can present long-term, industry-wide supply chain impact. Using CVE-2022-21894 as an example shows how such vulnerabilities can be exploited in the wild after one year, even with a vendor fix available.

Interestingly, CVE-2022-21894 reminds me of previous findings MS16-094/CVE-2016-3287 and MS16-100/CVE-2016-3320 (also known as Golden Key vulnerabilities) discovered by the same researcher Clark Zammis. The complexity of the modern Microsoft Windows Boot Manager (bootmgfw) grows with every new release of Windows, simultaneously expanding the attack surface. Modern UEFI-based Secure Boot schemes are extremely complicated to configure correctly and/or to reduce their attack surfaces meaningfully. That being said, bootloader attacks are not likely to disappear anytime soon.

A record number of high-impact vulnerabilities (228) were disclosed by the Binarly REsearch team in UEFI system firmware within one year. By design, most of these vulnerabilities bypass Secure Boot and allow attackers to persist at the firmware level.

The example of BlackLotus shows how old tricks with less complicated malware can be used to gain persistence below the operating system via exploitation of a known secure boot bypass vulnerability. Still, it is quite difficult to mitigate the BlackLotus/CVE-2022-21894 secure boot bypass across the industry. The UEFI Forum is usually responsible for coordinating and supporting the major vendor-independent mitigation of UEFI Revocation List Files (DBX). This list contains the hashes of signed bootloaders or firmware components that were blacklisted.

The problem with every blacklist technology is that it will always miss known problems if it is not updated or well maintained. In the history of UEFI Revocation List Files (DBX), Microsoft pushed mandatory updates only a handful of times. Other than OS vendors, who else should be responsible for these updates? Device vendors with firmware updates? As we all know, firmware updates typically occur only a few times a year, so any blacklist will be near useless with such minimal update frequency, but it is better than nothing.



We can see from the figure above that any compromised signed UEFI component can break secure boot integrity and bypass it. My mind immediately goes to the Binarly REsearch team's discovery of [BRLY-2021-003/CVE-2021-39297] (stack buffer overflow) vulnerability on HP devices last year (almost 8 months under disclosure process). Based on the demo below, an attacker can execute arbitrary code over HP Hardware Diagnostics UEFI application.

During its analysis of BlackLotus, the ESET research team discovered that the MokList NVRAM variable was modified. The MokList variable contains a list of authorized Machine Owner Keys (MOKs) and hashes (EFI_SIGNATURE_LIST according to the UEFI specification). But it's another example of a supply chain issue with broad secure boot implications in the field as the MokList variable can be modified in runtime. With the modified MokList variable, an attacker can easily load any self-signed shim bootloader, which is actually another vulnerability in the chain used to keep secure boot active.

In the modern secure boot, there are many inconsistencies due to many factors, including legacy compatibility issues. The Binarly REsearch Team has discussed these weaknesses for years at multiple public conferences, but the complexity of modern secure boot continues to increase.

Enterprise defenders and CISOs need to understand that threats below the operating system are clear and present dangers to their environments. Since this attack vector has significant benefits for the attacker, it is only going to get more sophisticated and complex. Vendor claims about security features can be completely opposite to the reality. Binarly's small research team was able to discover and disclose 228 high-impact vulnerabilities across all major enterprise vendors in a year, which we believe only scratches the surface. This research continues, on both sides of the fence (defense and offense).

A new name for old tricks

Are there any new techniques in the BlackLotus bootkit? My opinion is that BlackLotus is a good combination of well-known techniques. Proof of concept code for CVE-2022-21894 (public since August 2022) was taken from the GitHub repository of the researcher who found the vulnerability.

Binarly REsearch discovered new interesting data points about the nature of the bootkit code. It appears the author of the BlackLotus bootkit based their development on code from the Umap GitHub project (Windows UEFI bootkit that loads a generic driver manual mapper without using a UEFI runtime driver) or coincidently arrived at the same ideas. According to the first commit, Umap was released in April 2020.

This picture shows a comparison of the logic of the [main function]() from BlackLotus and Umap. Both look very similar and contain exactly the same steps with a few minor changes.

```
__int64 __fastcall sub_180002340(EFI_HANDLE ImageHandle, EFI_BOOT_SERVICES *gBS)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v3 = 0;
  WinloadHandle = 0i64;
  if ( (gBS->HandleProtocol)(ImageHandle, &EFI_LOADED_IMAGE_PROTOCOL_GUID, &EfiLoadedImageProtocolToHandle) >= 0 )
  {
    DecString = DecryptString(&unk_1800040C0, 0x1Fu, 1);// EFI\\Microsoft\\Boot\\winload.efi (replaced with bootmgr.efi)
    DevicePath = GetDevicePathByString(EfiLoadedImageProtocolToHandle->DeviceHandle, DecString);
    if ( DevicePath )
    {
      LOBYTE(BootPolicy) = 1;
      Status = (gBS->LoadImage)(BootPolicy, ImageHandle, DevicePath, 0i64, 0i64, &WinloadHandle);
      if ( Status >= 0
        && (gBS->OpenProtocol)(
             WinloadHandle,
             &EFI_LOADED_IMAGE_PROTOCOL_GUID,
             &OpenedEfiLoadedImageProtocol,
             ImageHandle,
             0i64,
             2) >= 0 )
      {
        BuildNumber = GetWindowsBuilldNumber(OpenedEfiLoadedImageProtocol->ImageBase);
        if ( BuildNumber >= 7600 )              // minimal build: 7600
        {
          ImageBase = OpenedEfiLoadedImageProtocol->ImageBase;
          ImageHeader = (ImageBase + ImageBase->e_lfanew);
          // search address inside ImgArchStartBootApplication
          // by pattern 41 b8 09 00 00 d0 (mov r8d, 0D0000009h)
          PatternAddress = SearchPatternInMemory(
                             ImageBase
                           + *(&ImageHeader->OptionalHeader.SizeOfUninitializedData
                             + ImageHeader->FileHeader.SizeOfOptionalHeader),
                             *(&ImageHeader->OptionalHeader.AddressOfEntryPoint
                             + ImageHeader->FileHeader.SizeOfOptionalHeader),
                             &gPatternInsideImgArchStartBootApplication,
                             &unk_180004048,
                             6u);
          if ( PatternAddress )
          {
            ImgArchStartBootApplicationAddress = GetAddressInMemory(
                                                   OpenedEfiLoadedImageProtocol->ImageBase,
                                                   PatternAddress);
            if ( ImgArchStartBootApplicationAddress )
            {
              Func = HookChainFunc;
              if ( BuildNumber < 9200 )
                Func = HookChainFuncorOlderWindows;
              Tpl = (gBS->RaiseTPL)(0x1Fi64);
              TrampolineHook(ImgArchStartBootApplicationAddress, Func, gImgArchStartBootApplicationOriginal);
              (gBS->RestoreTPL)(Tpl);
            }
          }
        }
      }
      (gBS->FreePool)(DevicePath);
      if ( Status >= 0 )
      {
        if ( (gBS->StartImage)(WinloadHandle, 0i64, 0i64) < 0 )
          (gBS->UnloadImage)(WinloadHandle);
        else
          return 1;
      }
    }
  }
  return v3;
}
```

The [routines responsible]() for installing the hook chain (start with *ImgArchStartBootApplication*) are very similar as shown in the figure.

```
  // search address inside ImgArchStartBootApplication
  // by pattern 41 b8 09 00 00 d0 (mov r8d, 0D0000009h)
  PatternAddress = SearchPatternInMemory(
                     ImageBase
                   + *(&ImageHeader->OptionalHeader.SizeOfUninitializedData
                     + ImageHeader->FileHeader.SizeOfOptionalHeader),
                     *(&ImageHeader->OptionalHeader.AddressOfEntryPoint
                     + ImageHeader->FileHeader.SizeOfOptionalHeader),
                     gPatternInsideImgArchStartBootApplication,
                     &unk_180004048,
                     6u);
  if ( PatternAddress )
  {
    ImgArchStartBootApplicationAddress = GetAddressInMemory(
                                           OpenedEfiLoadedImageProtocol->ImageBase,
                                           PatternAddress);
    if ( ImgArchStartBootApplicationAddress )
    {
      Func = HookChainFunc;
      if ( BuildNumber < 9200 )
        Func = HookChainFuncorOlderWindows;
      Tpl = (gBS->RaiseTPL)(0x1Fi64);
      TrampolineHook(ImgArchStartBootApplicationAddress, Func, gImgArchStartBootApplicationOriginal);
      (gBS->RestoreTPL)(Tpl);
    }
  }
}
```

BlackLotus trampoline code modification logic to setup hooks is identical to Umap code on GitHub.

```
void __fastcall TrampolineHook(__int64 dest, __int64 src, _BYTE *original)
{
  if ( dest && src )
  {
    if ( original )
      MemCopy(original, dest, 14i64);              // 14 -- JMP_SIZE
    MemCopy(dest, &gTrampolineHookData, 6i64);  // gTrampolineHookData = b"\xFF\x25\x00\x00\x00\x00"
    *(dest + 6) = src;
  }
}
```

```
VOID *TrampolineHook(VOID *dest, VOID *src, UINT8 original[JMP_SIZE]) {
    if (original) {
        MemCopy(original, src, JMP_SIZE);
    }

    MemCopy(src, "\xFF\x25\x00\x00\x00\x00", 6);
    *(VOID **)((UINT8 *)src + 6) = dest;

    return src;
}

VOID TrampolineUnHook(VOID *src, UINT8 original[JMP_SIZE]) {
    MemCopy(src, original, JMP_SIZE);
}
```

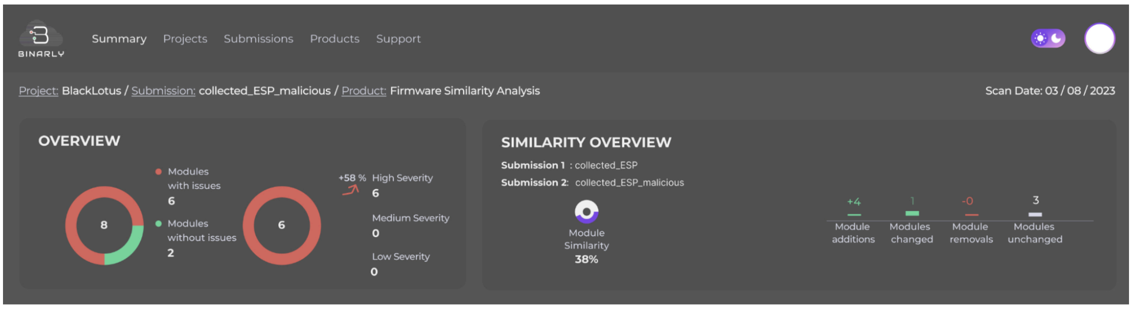
With these new data points, we can see how UEFI bootkit code reused from 2020 can be combined with publicly available proofs of concept for CVE-2022-1894 to lead to the creation of the BlackLotus bootkit. This relatively new secure boot bypass vulnerability (which the vendor claims is low-impact) has led to widespread distribution of old malicious UEFI bootkit code.

Zero-knowledge detection of new threats

Binarly has been focused from the beginning on developing proactive technology based on deep code inspection to detect unknown threats and vulnerabilities to help the industry recover from repeatable failures. The Binarly

Platform dashboard below shows code similarity proactive detection based on machine learning models guided by code-based embeddings.

Using the BlackLotus components, we simulated infection on one of the machines available in our lab. Binarly Platform was used to compare collected snapshots of the infected and clean EFI System Partitions (ESP). Based on function similarity, we detect the replacement of bootmgfw.efi with the shim.



The BlackLotus anomaly was proactively detected with zero knowledge about this threat, and the explained code similarity failures make it actionable for the security and incident response teams to conduct further investigation.

Let's explore the code similarity detection in more detail. The figure below shows the detailed output for the detected anomalies from the BlackLotus malicious components added to ESP partition.

```
[
  {
    "module_similarity": "Very dissimilar (less than 20% similarity)",
    "module_additions": 4,
    "module_removals": 0,
    "modules_new_or_changed": 5,
    "modules_unchanged": 3,
    "module_scan_results": [
      {
        "module": {
          "name": "ESP:\\system32\\bootmgr.efi",
          "hash": "971d3d12b577a942048c7d3302b8d183b111eaf1941f5ba84d9f0792e5fe6ebd"
        },
        "checks": [
          {
            "name": "NewModule",
            "meta": {
              "description": "Check if the module has a path that is not present in a previous snapshot version.",
              "severity": 2
            },
            "status": 1
          }
        ]
      },
      {
        "module": {
          "name": "ESP:\\EFI\\Microsoft\\Boot\\grubx64.efi",
          "hash": "fb07b7ae72ced82f0b551becb384f98d12aae1690cc4667180920f7649a543be"
        },
        "checks": [
          {
            "name": "NewModule",
            "meta": {
              "description": "Check if the module has a path that is not present in a previous snapshot version.",
              "severity": 2
            },
            "status": 1
          }
        ]
      },
      {
        "module": {
          "name": "ESP:\\system32\\hvloader.efi",
          "hash": "a9ca98bd4664f2d206ac1bc2db7a4af92cc6e33aff96e10658b9383364a33feb"
        },
        "checks": [
          {
            "name": "NewModule",
            "meta": {
              "description": "Check if the module has a path that is not present in a previous snapshot version.",
              "severity": 2
            },
            "status": 1
          }
        ]
      },
      {
        "module": {
          "name": "ESP:\\EFI\\Microsoft\\Boot\\winload.efi",
          "hash": "490c1ccec40dbc79a8575d2a3ad8124c03ea56d609be2acdb6dd7bfca0ea2924"
        },
        "checks": [
          {
            "name": "NewModule",
            "meta": {
              "description": "Check if the module has a path that is not present in a previous snapshot version.",
              "severity": 2
            },
            "status": 1
          }
        ]
      },
```


Changes Summary

❶ Module Addition

❷ Module Addition

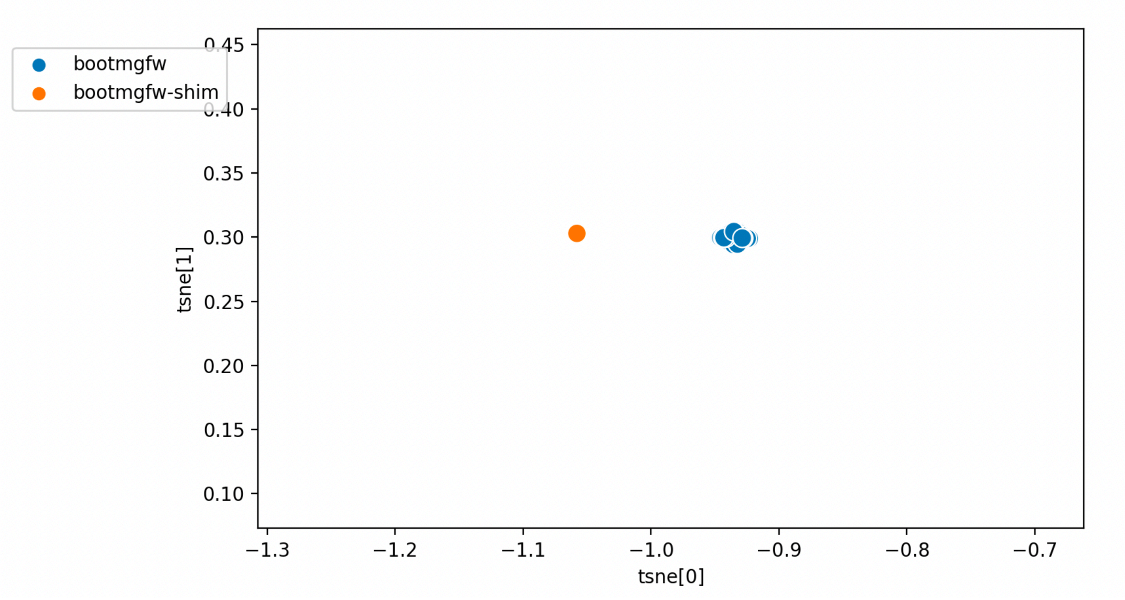
❸ Module Addition

❹ Module Addition

Code similarity detects anomalies based on code changes and integrity checking heuristics as shown in the figure below.

```
{
  "module": {
    "name": "ESP:\\EFI\\Microsoft\\Boot\\bootmgfw.efi",
    "hash": "e8cdc0697748e702cf2916a2c5670325a891402ee38c98d91873a0f03e3f9025"
  },
  "checks": [    ⑤ Integrity Failed
    {
      "name": "ChangedModuleHash",
      "meta": {
        "description": "Check if the module has a hash that is different from the same module in a previous
            snapshot version.",
        "extra_info": {
          "modules": [
            {
              "hash": "490c1ccec40dbc79a8575d2a3ad8124c03ea56d609be2acdb6dd7bfca0ea2924",
              "name": "ESP:\\EFI\\Microsoft\\Boot\\bootmgfw.efi"
            }
          ]
        },
        "severity": 2
      },
      "status": 1
    },      ⑥ Code Changed
    {
      "name": "FunctionSimilarity",
      "meta": {
        "description": "Check how similar the module's functions are to the same module in a previous snapshot
            version.",
        "extra_info": {
          "modules": [
            {
              "hash": "490c1ccec40dbc79a8575d2a3ad8124c03ea56d609be2acdb6dd7bfca0ea2924",
              "name": "ESP:\\EFI\\Microsoft\\Boot\\bootmgfw.efi",
              "similarity": "Very dissimilar (less than 20% similarity)"
            }
          ]
        },
        "severity": 2
      },
      "status": 1
    }
```

Here is a visual explanation of the outlier (compromised ESP partition) and details of the algorithm to detect anomaly:



Binarly's Platform is able to analyze modules and executables based on semantic similarity. The figure above shows a visualization of the program embeddings for bootmgfw and the shim. For code function signatures, we used data clustering algorithm DBSCAN with a precomputed Gower distance to detect anomalies. This is where we detect that additional suspicious modules were added. Afterwards, we visualized the results using the T-SNE algorithm.

What about detection based on FwHunt?

We continue to maintain the public FwHunt rules database and today we released a new semantic-based rule to cover malicious bootloader components from the BlackLotus bootkit.

BlackLotus rule is included in FwHunt's GitHub repository. To use these rules you will need FwHunt Community Scanner (fwhunt-scan).

We need to increase the industry awareness to firmware related threats and build more effective threat hunting programs with cross-industry collaboration between the vendors to mutually benefit customers and provide better detection rates.

Source: https://www.binarly.io/posts/The_Untold_Story_of_the_BlackLotus_UEFI_Bootkit/index.html