Bootkitty: Analyzing the first UEFI bootkit for Linux

By Martin SmolárPeter Strýček

Archived: 2026-04-05 17:09:25 UTC

> UPDATE (December 2nd, 2024): The bootkit described in this report seems to be part of a project created by cybersecurity students participating in Korea's Best of the Best (BoB) training program. As they informed us: "*The primary aim of this project is to raise awareness within the security community about potential risks and to encourage proactive measures to prevent similar threats. Unfortunately, few bootkit samples were disclosed prior to the planned conference presentation.*" This supports our belief that it was an initial proof of concept rather than production-ready malware used by real threat actors. Nonetheless, the blog post remains accurate – it is a functional bootkit with limited support and represents the first UEFI bootkit proof of concept for Linux OS.

Over the past few years, the UEFI threat landscape, particularly that of UEFI bootkits, has evolved significantly. It all started with the first UEFI bootkit proof of concept (PoC) described by Andrea Allievi in 2012, which served as a demonstration of deploying bootkits on modern UEFI-based Windows systems, and was followed with many other PoCs (EfiGuard, Boot Backdoor, UEFI-bootkit). It took several years until the first two real UEFI bootkits were discovered in the wild (ESPecter, 2021 ESET; FinSpy bootkit, 2021 Kaspersky), and it took two more years until the infamous BlackLotus – the first UEFI bootkit capable of bypassing UEFI Secure Boot on up-to-date systems – appeared (2023, ESET).

A common thread among these publicly known bootkits was their exclusive targeting of Windows systems. Today, we unveil our latest discovery: the first UEFI bootkit designed for Linux systems, named Bootkitty by its creators. We believe this bootkit is merely an initial proof of concept, and based on our telemetry, it has not been deployed in the wild. That said, its existence underscores an important message: UEFI bootkits are no longer confined to Windows systems alone.

The bootkit's main goal is to disable the kernel's signature verification feature and to preload two as yet unknown ELF binaries via the Linux init process (which is the first process executed by the Linux kernel during system startup). During our analysis, we discovered a possibly related unsigned kernel module – with signs suggesting that it could have been developed by the same author(s) as the bootkit – that deploys an ELF binary responsible for loading yet another kernel module unknown during our analysis.

> **Key points of this blogpost:**
>
> - In November 2024, a previously unknown UEFI application, named bootkit.efi, was uploaded to VirusTotal.
> - Our initial analysis confirmed it is a UEFI bootkit, named Bootkitty by its creators and surprisingly the first UEFI bootkit targeting Linux, specifically, a few Ubuntu versions.
> - Bootkitty is signed by a self-signed certificate, thus is not capable of running on systems with UEFI Secure Boot enabled unless the attackers certificates have been installed.

> - Bootkitty is designed to boot the Linux kernel seamlessly, whether UEFI Secure Boot is enabled or not, as it patches, in memory, the necessary functions responsible for integrity verification before GRUB is executed.
> - bootkit.efi contains many artifacts suggesting this is more like a proof of concept than the work of an active threat actor.
> - We discovered a possibly related kernel module, which we named BCDropper, that deploys an ELF program responsible for loading another kernel module.

Bootkitty overview

As mentioned in the introduction, Bootkitty contains many artifacts suggesting that we might be dealing with a proof of concept instead of actively used malware. In this section, we look more closely at these artifacts, plus other basic information about the bootkit.

Bootkitty contains two unused functions, capable of printing special strings to the screen during its execution. The first function, whose output is depicted in Figure 1, can print ASCII art that we believe represents a possible name of the bootkit: Bootkitty.

Figure 1. ASCII art embedded in the bootkit

The second function, can print text, shown in Figure 2, containing the list of possible bootkit authors and other persons that perhaps somehow participated in its development. One of the names mentioned in the image can be found on GitHub, but the profile does not have any public repository that would contain or mention a UEFI bootkit project; therefore, we can neither confirm nor deny authenticity of the names mentioned in the bootkit.

Figure 2. List of names embedded in the bootkit (redacted)

During every boot, Bootkitty prints on screen the strings shown in Figure 3.

Figure 3. Bootkitty's welcome message

Note that the BlackCat name is referenced also in the loadable kernel module described later. Despite the name, we believe there is no connection to the ALPHV/BlackCat ransomware group. This is because BlackCat is a name used by researchers and Bootkitty was developed in C, while the group calls itself ALPHV and develops its malware exclusively in Rust.

As mentioned earlier, Bootkitty currently supports only a limited number of systems. The reason is that to find the functions it wants to modify in memory, it uses hardcoded byte patterns. While byte-pattern matching is a common technique when it comes to bootkits, the authors didn't use the best patterns for covering multiple kernel or GRUB versions; therefore, the bootkit is fully functional only for a limited number of configurations. What limits the use of the bootkit even more is the way it patches the decompressed Linux kernel: as shown in Figure 4, once the kernel image is decompressed, Bootkitty simply copies the malicious patches to the hardcoded offsets within the kernel image.

```
kernel_image = gDecompressedKernel;
*(gDecompressedKernel + 0x19AC5E0) = '1BoB';
*(kernel_image + 0x19AC5E4) = '3';
qmemcpy((kernel_image + 0x29E84A7), "BoB13", 5);
result = v2;
*(kernel_image + 0x3FA4B5) = 0xB8E5894855LL;
*(kernel_image + 0x3FA4BD) = 0xC35D00;
*(kernel_image + 0x260ED20) = 0xFFFFFFFF82505000uLL;
strcpy((kernel_image + 0x1705000), "LD_PRELOAD=/opt/injector.so /init");
```

Figure 4. Bootkitty's code responsible for patching the decompressed kernel before it is executed

We explain how the bootkit gets to the actual kernel patching later in the *Linux kernel image decompression hook* section; for now, just note that due to the lack of kernel-version checks in the function shown in Figure 4, Bootkitty can get to the point where it patches completely random code or data at these hardcoded offsets, thus crashing the system instead of compromising it. This is one of the facts that supports proof of concept. On the other hand, it might be an initial not-production-ready version of malware created by malicious threat actors.

Last but not least, the bootkit binary is signed by the self-signed certificate shown in Figure 5.

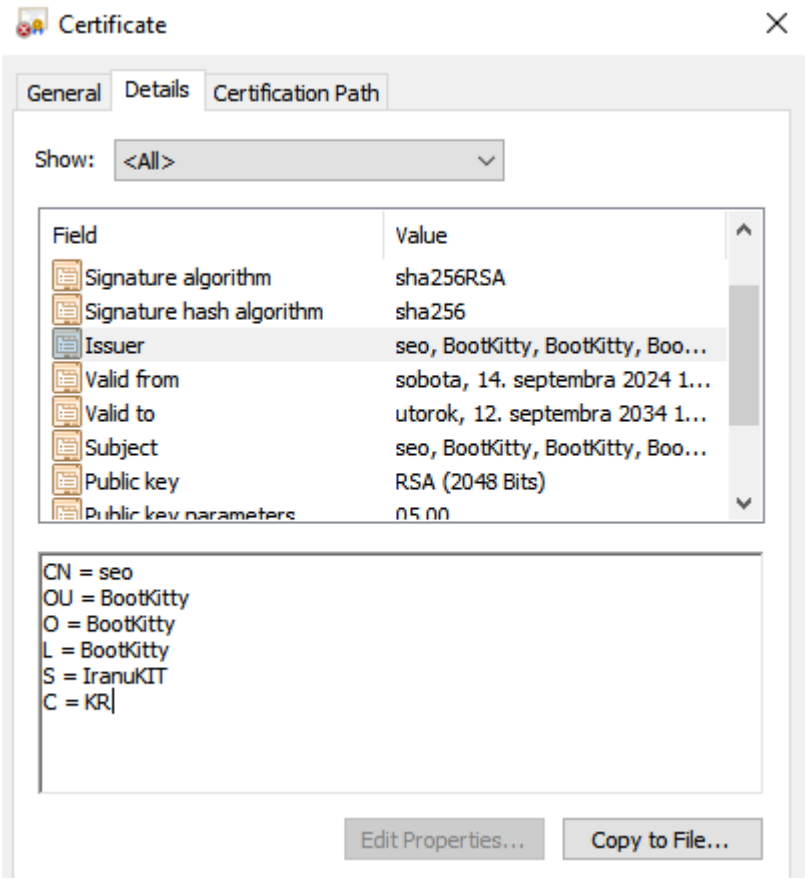

Figure 5. Self-signed certificate used to sign the bootkit

Techical analysis

We start with an overview of Bootkitty's execution, as depicted in Figure 6. First, we briefly describe the main functionality and then in subsequent sections we go into more details.

There are three main parts we focus on:

- Execution of the bootkit and patching of the legitimate GRUB bootloader (points 4 and 5 in Figure 6).
- Patching of the Linux kernel's EFI stub loader (points 6 and 7 in Figure 6).
- Patching of the decompressed Linux kernel image (points 8 and 9 in Figure 6).

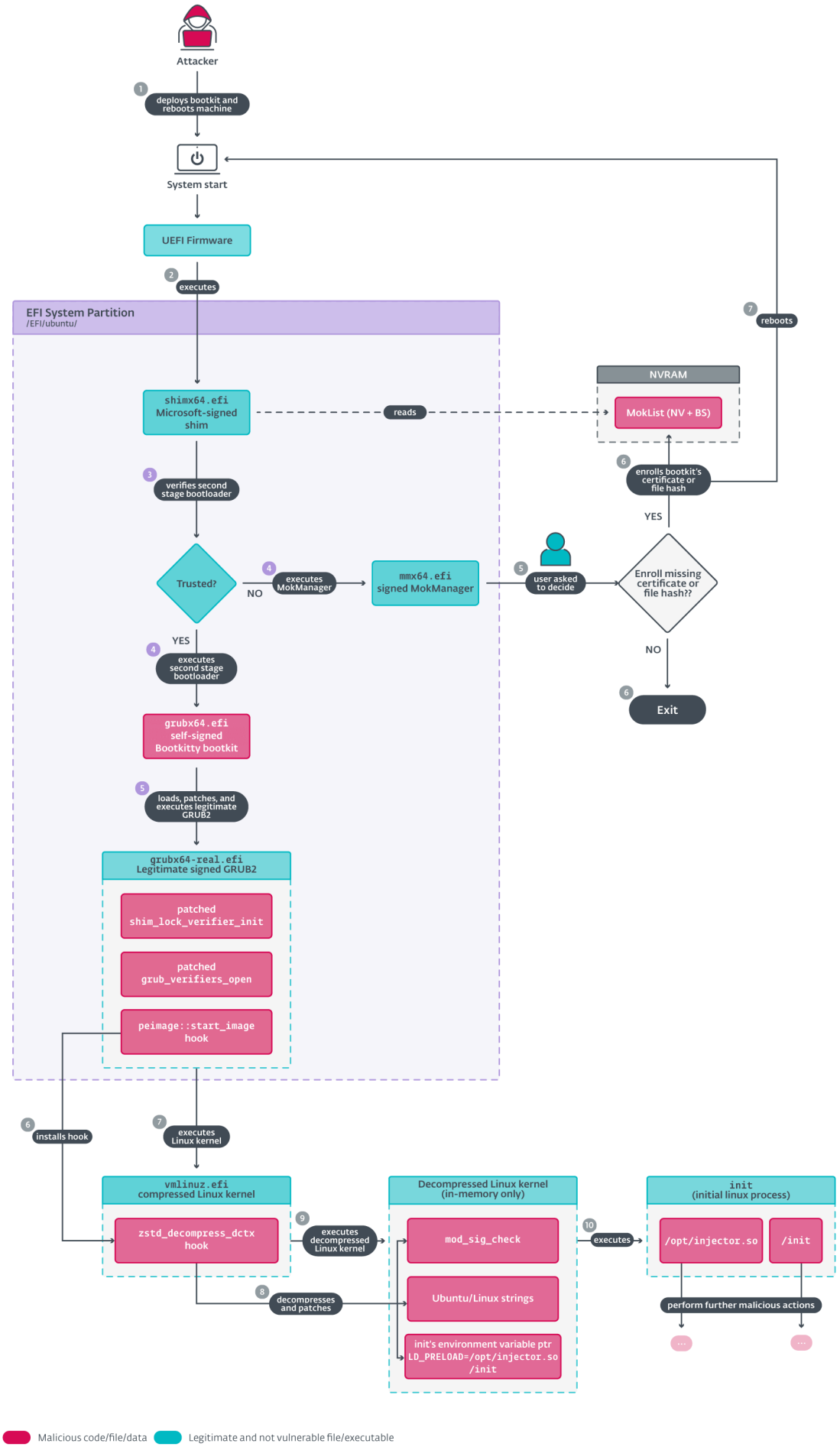
Attacker
1
deploys bootkit and reboots machine
System start
UEFI Firmware
2
executes
EFI System Partition
/EFI/ubuntu/
shimx64.efi
Microsoft-signed shim
reads
NVRAM
MokList (NV + BS)
7
reboots
3
verifies second stage bootloader
6
enrolls bootkit's certificate or file hash
YES
Trusted?
4
executes MokManager
NO
mmx64.efi
signed MokManager
5
user asked to decide
Enroll missing certificate or file hash??
NO
6
Exit
YES
4
executes second stage bootloader
grubx64.efi
self-signed
Bootkitty bootkit
5
loads, patches, and executes legitimate GRUB2
grubx64-real.efi
Legitimate signed GRUB2
patched shim_lock_verifier_init
patched grub_verifiers_open
peimage::start_image hook
6
installs hook
7
executes Linux kernel
vmlinuz.efi
compressed Linux kernel
zstd_decompress_dctx hook
9
executes decompressed Linux kernel
Decompressed Linux kernel
(in-memory only)
mod_sig_check
10
executes
init
(initial linux process)
/opt/injector.so
/init
8
decompresses and patches
Ubuntu/Linux strings
perform further malicious actions
init's environment variable ptr
LD_PRELOAD=/opt/injector.so
/init
Malicious code/file/data
Legitimate and not vulnerable file/executable

Figure 6. Bootkitty execution overview

Initialization and GRUB hooking

After Bootkitty is executed by the shim, it checks to see whether UEFI Secure Boot is enabled by examining the value of the SecureBoot UEFI variable, and proceeds to hook two functions from the UEFI authentication protocols if so (this process is shown in Figure 7):

- EFI_SECURITY2_ARCH_PROTOCOL.FileAuthentication: this function is used by the firmware to measure and verify the integrity of UEFI PE images. Bootkitty's hook function modifies the output of this function so that it always returns EFI_SUCCESS, meaning that the verification succeeded.
- EFI_SECURITY_ARCH_PROTOCOL.FileAuthenticationState: this function is used by the firmware to execute a platform-specific policy in response to different authentication status values. Again, the bootkit's hook modifies it in a way that it always returns EFI_SUCCESS, meaning that the firmware can use the file regardless of its actual authentication status.

```
if ( !(gST_1->RuntimeServices->GetVariable)(
        L"SecureBoot",
        &EFI_SIMPLE_BOOT_FLAG_VARIABLE_GUID,
        0LL,
        &LoadedImage,
        &SecBootStatus) )
{
  if ( SecBootStatus )
  {
    sp_override = security_policy_mok_override;
    sp_allow = security_policy_mok_allow;
    sp_deny = security_policy_mok_deny;
    Handle = 0LL;
    if ( !gOrigFileAuthenticationState )
    {
      (gST_1->BootServices->LocateProtocol)(&EFI_SECURITY2_ARCH_PROTOCOL_GUID, 0LL, &Handle);
      if ( !(gST_1->BootServices->LocateProtocol)(&EFI_SECURITY_ARCH_PROTOCOL_GUID, 0LL, &v47) )
      {
        if ( Handle )
        {
          es2fa = Handle->FileAuthentication;
          Handle->FileAuthentication = Hook_S2FileAuthentication;
        }
        gOrigFileAuthenticationState = v47->FileAuthenticationState;
        v47->FileAuthenticationState = Hook_FileAuthenticationState;
      }
    }
  }
}
```

Figure 7. Hooking of the UEFI security authentication protocols

After checking the status of UEFI Secure Boot, Bootkitty proceeds to load the legitimate GRUB from the hardcoded path on the EFI system partition: /EFI/ubuntu/grubx64-real.efi. This file should be a backup, created by the attacker, of a legitimate GRUB. Once GRUB is loaded (not yet executed), the bootkit starts patching and hooking the following code in GRUB's memory:

- The start_image function within the peimage GRUB module (a module embedded inside GRUB). This function is responsible for starting an already loaded PE image, and it's invoked by GRUB to start the Linux kernel's EFI stub binary (known in general as vmlinuz.efi or vmlinuz). The hook function takes advantage of the fact that at the moment the hook is executed, vmlinuz is already loaded into memory (but

hasn't been executed yet), and patches the function responsible for decompressing the actual Linux kernel image inside vmlinuz (note that in some cases, due to the way the Linux kernel is compiled, it can be quite challenging to find the exact name of the function being patched; however, we believe that this time it should be the zstd_decompress_dctx function). More details about the decompression hook are in the *Linux kernel image decompression hook* section.

- The shim_lock_verifier_init function, which is part of the shim_lock verifier mechanism inside GRUB – this should be activated automatically if UEFI Secure Boot is enabled. It is responsible for deciding whether the files provided (e.g., GRUB modules, Linux kernel, configurations…) should be verified or not during the boot. The installed hook, however, is somehow confusing and the author's intentions are unclear because it modifies shim_lock_verifier_init's output in a way that it sets the output flag to GRUB_VERIFY_FLAGS_SINGLE_CHUNK (value 2) for any file type provided, which should, according to the GRUB manual, strengthen the security even more. Interestingly, due to the hook described in the next point, this shim_lock_verifier_init function is not even called during the boot, thus becoming irrelevant.
- The grub_verifiers_open function. This function is invoked by GRUB anytime it opens a file, and is responsible for checking whether the installed GRUB file verifiers (this includes the shim_lock verifier described above) require integrity verification for the file being loaded. The function is hooked by the bootkit in a way that it returns immediately without proceeding to any signature checks (note that this means that it does not even execute the previously hooked shim_lock_verifier_init function).

Linux kernel image decompression hook

This hook is responsible for patching the decompressed Linux kernel image. The hook is called right before the kernel image is decompressed, so the hook restores the original decompression function's bytes and executes the original function to decompress the kernel image before proceeding to the kernel patching.

Now, as the kernel is decompressed and lies in the memory untouched (still hasn't been executed), the hook code patches it at hardcoded offsets (in memory only). Specifically, as shown in Figure 8, it:

- Rewrites the kernel version and Linux banner strings with the text BoB13 (this has no significant impact on the system).
- Hooks the module_sig_check function.
- Patches pointer/address to the first environment variable of the init process.

```
UINTN __fastcall hook_vmlinuz_decompress(void *a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  OrigDecompress = g_vmlinuz_decompress_patt_addr;
  // restore original function's bytes
  *g_vmlinuz_decompress_patt_addr = gOrig_vmlinuz_decompress_bytes;
  *(OrigDecompress + 8) = gOrig_vmlinuz_decompress_bytes_0;
  *(OrigDecompress + 9) = gOrig_vmlinuz_decompress_bytes_0_0;
  *(OrigDecompress + 10) = gOrig_vmlinuz_decompress_bytes_0_0_0;
  *(OrigDecompress + 11) = gOrig_vmlinuz_decompress_bytes_0_0_1;

  // // call the original decompress function to decompress the kernel image
  (OrigDecompress)();
  v2 = sub_18000C88A(a1);
  sub_18000DF00(a1, v2, v3);
  v8 = sub_18000C88E(a1, v2, v4, v5, v6, v7, v15);
  sub_18000DF00(v8, v2, v9);
  v10 = sub_18000C894();
  sub_18000DF00(v8, v2, v11);
  gDecompressedKernel = v8 | (v10 << 32);
  sub_18000C89A();
  sub_18000DF00(v8, v2, v12);
  kernel_image = gDecompressedKernel;
  *(gDecompressedKernel + 0x19AC5E0) = '1BoB';
  *(kernel_image + 0x19AC5E4) = '3';
  qmemcpy((kernel_image + 0x29E84A7), "BoB13", 5);
  result = v2;
  *(kernel_image + 0x3FA4B5) = 0xB8E5894855LL;
  *(kernel_image + 0x3FA4BD) = 0xC35D00;
  *(kernel_image + 0x260ED20) = 0xFFFFFFFF82505000uLL;
  strcpy((kernel_image + 0x1705000), "LD_PRELOAD=/opt/injector.so /init");
  return result;
}
```

Figure 8. Bootkitty's kernel-decompression hook inside vmlinuz

The function module_sig_check is patched to always return 0. This function is responsible for checking whether the module is validly signed. By patching the function to return 0, the kernel will load any module without verifying the signature. On Linux systems with UEFI Secure Boot enabled, kernel modules *need* to be signed if they are meant to be loaded. This is also the case when the kernel is built with CONFIG_MODULE_SIG_FORCE enabled or when module.sig_enforce=1 is passed as a kernel command line argument, as described in the Linux kernel documentation. The likely scenario is that at least one malicious kernel module is loaded at a later phase, such as the dropper analyzed below.

The first process that the Linux kernel executes is init from the first hardcoded path that works (starting with /init from initramfs), along with command line arguments and environment variables. The hook code replaces the first environment variable with LD_PRELOAD=/opt/injector.so /init. LD_PRELOAD is an environment variable that is used to load ELF shared objects before others and can be used to override functions. It is a common technique used by attackers to load malicious binaries. In this case, the /opt/injector.so and /init ELF shared objects are loaded when the init process starts. This is where the intention becomes less clear, mainly why the second string /init is part of LD_PRELOAD.

We have not discovered any of these possibly malicious ELF shared objects, although just as this blogpost was being finalized for publication, a write-up describing the missing components mentioned in our report has been published. Now it's clear they are used just to load another stage.

Impact and remediation

Apart from loading unknown ELF shared objects, Bootkitty leaves footprints in the system. The first is the intended, albeit not necessary, modification of kernel version and Linux banner strings. The former can be seen by running uname -v (Figure 9) and the latter by running dmesg (Figure 10).

```
$ uname -v
#44-BoB13u SMP PREEMPT_DYNAMIC Tue Aug 13 13:35:26 UTC 2024
```

Figure 9. BoB13 *string in uname output*

```
# dmesg
[    0.000000] BoB13 version 6.8.0-44-generic (buildd@lcy02-amd64-08
2) (x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-23ubuntu4) 13.2.0, GNU ld
 (GNU Binutils for Ubuntu) 2.42) #44-Ubuntu SMP PREEMPT_DYNAMIC Tue
Aug 13 13:35:26 UTC 2024 (Ubuntu 6.8.0-44.44-generic 6.8.12)
```

Figure 10. BoB13 *string in dmesg output*

During our analysis, the output of the command dmesg also included details about how the init process was run. As depicted in Figure 11, the process was run with the LD_PRELOAD environment variable (it was originally HOME=/ and was replaced with LD_PRELOAD=/opt/injector.so /init by the bootkit).

```
Run /init as init process
  with arguments:
    /init
    splash
  with environment:
    LD_PRELOAD=/opt/injector.so /init
    TERM=linux
    BOOT_IMAGE=/boot/vmlinuz-6.8.0-44-generic
```

Figure 11. init *process arguments and environment variables in dmesg output*

Note in Figure 11 that the word /init in the first line corresponds to the legitimate program in initramfs that eventually passes control to systemd on default Ubuntu installations. The presence of the LD_PRELOAD environment variable can also be verified by inspecting the file /proc/1/environ.

After booting up a system with Bootkitty in our testing environment, we noticed that the kernel was marked as tainted (command from Figure 12 can be used to check the tainted value), which was not the case when the bootkit was absent. Another way to tell whether the bootkit is present on the system with UEFI Secure Boot enabled is by attempting to load an unsigned dummy kernel module during runtime. If it's present, the module will be loaded; if not – the kernel refuses to load it.

```
# cat /proc/sys/kernel/tainted
8192
```

Figure 12. Tainted state right after the system has started with Bootkitty

A simple remedy tip to get rid of the bootkit is to move the legitimate /EFI/ubuntu/grubx64-real.efi file back to its original location, which is /EFI/ubuntu/grubx64.efi. This will make shim execute the legitimate GRUB and thus the system will boot up without the bootkit (note that this covers only the scenario when the bootkit is deployed as /EFI/ubuntu/grubx64.efi).

BCDropper and BCObserver

In addition to the bootkit, we discovered a possibly related unsigned kernel module we named BCDropper, uploaded to VirusTotal around the same time and by the same submitter's ID as the bootkit, containing hints that it might have been developed by the same author as the bootkit, such as:

- a BlackCat string in the output of the modinfo command's output, shown in Figure 13,
- another presence of the blackcat string in the debug paths in the module's binary, shown in Figure 14, and
- it contains an unused file-hiding function that hides specific entries from directory listings. As shown in Figure 15, one of the hardcoded filename string prefixes used to filter-out these entries is injector (note that Bootkitty tries to preload a shared-library from the path /opt/injector.so)

However, even with the evidence presented, we cannot say for sure whether or not the kernel module is related to Bootkitty (or was created by the same developer). Also, the kernel version mentioned in Figure 13 (6.8.0-48-generic) is not supported by the bootkit.

```
$ modinfo dropper.ko
filename:       /home/ubuntu/dropper.ko
description:    Dropper
author:         BlackCat
license:        GPL
srcversion:     496CD0E590D503DCA42AE18
depends:
retpoline:      Y
name:           dropper
vermagic:       6.8.0-48-generic SMP preempt mod_unload modversions
```

Figure 13. Dropper module information

```
/home/blackcat/Desktop/workspace/Rootkit/rootkit_loader/dropper/dropper.c
ric /home/blackcat/Desktop/workspace/Rootkit/rootkit_loader/dropper /home/
ootkit_loader/dropper/../Utils ./include/linux ./arch/x86/include/asm ./in
```

Figure 14. Dropper debug symbols referencing blackcat

```
hidden_files    dq offset aObserver
; DATA XREF: hook_getdents64:loc_757↑r
; hook_getdents:loc_B27↑r; "observer"
                dq offset aDropper; "dropper"
                dq offset aRootkit; "rootkit"
                dq offset aInjector; "injector"
_data           ends
```

Figure 15. List of files, in the dropper, to hide

As its name suggests, the kernel module drops an embedded ELF file we named BCObserver, specifically to /opt/observer, and executes it via /bin/bash (Figure 17). On top of that, the module hides itself by removing its entry from the module list. The kernel module also implements other rootkit-related functionalities like hiding files (those in Figure 15), processes, and open ports, but they are not directly used by the dropper.

```
if ( drop_file(observer_data, 0x3FE0uLL, "/opt/observer") )
    return -1;
argv[0] = "/bin/bash";
v1 = "/bin/bash";
argv[1] = "-c";
argv[2] = "/opt/observer";
argv[3] = 0LL;
envp[0] = "HOME=/";
envp[1] = "PATH=/sbin:/usr/sbin:/bin:/usr/bin";
envp[2] = 0LL;
while ( call_usermodehelper(v1, argv, envp, 1LL) )
    v1 = argv[0];
hide_module();
```

Figure 16. Hex-Rays decompiled dropper code

BCObserver is a rather simple application that waits until the display manager gdm3 is running, and then loads an unknown kernel module from /opt/rootkit_loader.ko via the finit_module system call. By waiting for the display manager to start, the code ensures that the kernel module is loaded after the system is fully booted up.

```
while ( !is_gdm3_running() )
    ;
load_module_from_path("/opt/rootkit_loader.ko");
```

Figure 17. Hex-Rays decompiled observer code

While we cannot confirm whether the dropper is somehow related to the bootkit, and if so, how it is meant to be executed, we're quite sure that the bootkit patches the module_sig_check function for a reason, and loading an unsigned kernel module (such as the dropper described here) would definitely make sense.

Conclusion

Whether a proof of concept or not, Bootkitty marks an interesting move forward in the UEFI threat landscape, breaking the belief about modern UEFI bootkits being Windows-exclusive threats. Even though the current version from VirusTotal does not, at the moment, represent a real threat to the majority of Linux systems, it emphasizes the necessity of being prepared for potential future threats.

To keep your Linux systems safe from such threats, make sure that UEFI Secure Boot is enabled, your system firmware and OS are up-to-date, and so is your UEFI revocations list.

> *For any inquiries about our research published on WeLiveSecurity, please contact us at threatintel@eset.com.*
>
> *ESET Research offers private APT intelligence reports and data feeds. For any inquiries about this service, visit the ESET Threat Intelligence page.*

IoCs

A comprehensive list of indicators of compromise (IoCs) and samples can be found in our GitHub repository.

Files

SHA-1	Filename	Detection	Description
35ADF3AED60440DA7B80 F3C452047079E54364C1	bootkit.efi	EFI/Agent.A	Bootkitty UEFI bootkit.
BDDF2A7B3152942D3A82 9E63C03C7427F038B86D	dropper.ko	Linux/Rootkit.Agent.FM	BCDropper.
E8AF4ED17F293665136E 17612D856FA62F96702D	observer	Linux/Rootkit.Agent.FM	BCObserver.

MITRE ATT&CK techniques

This table was built using version 16 of the MITRE ATT&CK framework.

Tactic	ID	Name	Description
Resource Development	T1587.001	Develop Capabilities: Malware	Bootkitty is a brand-new UEFI bootkit developed by an unknown author.
	T1587.002	Develop Capabilities: Code Signing Certificates	Bootkitty sample is signed with a self-signed certificate.
Execution	T1106	Native API	BCObserver uses the finit_module system call to load a kernel module.

Tactic	ID	Name	Description
	T1129	Shared Modules	Bootkitty uses LD_PRELOAD to preload shared modules from a hardcoded path into the init process during system start.
Persistence	T1574.006	Hijack Execution Flow: Dynamic Linker Hijacking	Bootkitty patches init's environment variable with LD_PRELOAD so it loads a next stage when executed.
	T1542.003	Pre-OS Boot: Bootkit	Bootkitty is a UEFI bootkit meant to be deployed on the EFI System Partition.
Defense Evasion	T1014	Rootkit	BCDropper serves as a rootkit implemented as a loadable kernel module for Linux systems.
	T1562	Impair Defenses	Bootkitty disables signature verification features in the GRUB bootloader and Linux kernel.
	T1564	Hide Artifacts	BCDropper hides itself by removing its module's entry from the kernel's modules list.

Source: https://www.welivesecurity.com/en/eset-research/bootkitty-analyzing-first-uefi-bootkit-linux/